# Contrôle d'accès obligatoire pour systèmes à objets : défense en profondeur des objets Java

Benjamin Venelle

# UNIVERSITÉ D'ORLÉANS

***ÉCOLE DOCTORALE***
***MATHEMATIQUES, INFORMATIQUE, PHYSIQUE THEORIQUE***
***ET INGENIERIE DES SYSTEMES***

Laboratoire d'Informatique Fondamentale d'Orléans (LIFO)
Equipe Sécurité des Systèmes Distribués (SDS)

## THÈSE

Présentée par :

**Benjamin VENELLE**

Soutenue le : **16 juillet 2015**

Pour obtenir le grade de : **Docteur de l'université d'Orléans**
Discipline / Spécialité : **informatique**

# Contrôle d'accès obligatoire pour systèmes à objets

# Défense en profondeur des objets Java

**THÈSE dirigée par :**
**Christian TOINARD** Professeur des Universités (INSA Centre Val de Loire)

**RAPPORTEURS :**
**Franck BARBIER** Professeur des Universités (Université de Pau)
**Damien SAUVERON** Maitre de Conférences - HDR (Université de Limoges)

---

**JURY :**
**Jean-Michel COUVREUR** Professeur des Universités (Université d'Orléans)
**Hong-Yon LACH** Ingénieur recherche (Alcatel-Lucent Bell Labs France)

# Remerciements

Je tiens à remercier mon ami et mentor Christian TOINARD pour son accompagnement, ses critiques constructives et surtout son soutien indéfectible dans les moments de doutes. Je garderai à jamais le souvenir de nos stimulants échanges sur les thèmes de la recherche scientifique, la place du chercheur et bien d'autres choses encore. Je saurai faire honneur à toutes les connaissances et réflexions que Christian a su me transmettre.

Je souhaite également rendre hommage à mon ami Bertrand MARQUET sans qui cette thèse n'aurait eu lieu. Dès le départ Bertrand m'a accordé sa confiance dans la réussite de ce projet et a su convaincre ses supérieurs de l'intérêt de ce sujet. Merci.

Je remercie aussi mes collègues de travail avec lesquels j'ai eu de fructueux échanges et qui m'ont aidé autant qu'ils le pouvaient. A Messieurs Arnaud ANSIAUX, Haithem EL ABED, Samuel DUBUS, Sylvain L'HARIDON, Waël KANOUN, Hong-Yon LACH, Antony MARTIN, Serge PAPILLON, Abdullatif SHIKFA ainsi qu'à Madame Layal EL SAMARJI : merci.

Je souhaite aussi remercier Messieurs Franck BARBIER et Damien SAUVERON qui m'ont fait l'honneur d'évaluer mon travail.

Plus globalement, je souhaite remercier le LIFO et son directeur Jérôme DURAND-LOSE ainsi que Alcatel-Lucent Bell Labs France et son président Jean-Luc BEYLAT pour m'avoir accueilli au sein de leur organisation respective et m'avoir fourni les moyens nécessaires à la réussite de cette thèse.

Enfin, je ne saurais oublier mes proches, famille comme amis, qui m'ont prodigué les encouragements dont j'avais besoin et tout le soutien sans lequel je n'aurais pu finaliser cette thèse.

*A la mémoire de Claude VENELLE, disparu le* $1^{er}$ *novembre 2013.*

# Table des matières

# Table des figures

Chapitre 1

# Introduction

## Sommaire



> *Pour qu'une technologie réussisse, la réalité doit précéder sur les relations publiques, car la nature ne peut pas être dupée.*
>
> Richard P. Feynman [1]

Lorsqu'une entreprise s'engage contractuellement à fournir des équipements et services conformes aux spécifications établies, le déploiement d'un correctif de sécurité prend du temps et engendre des coûts [2]. Quand le nombre de vulnérabilités critiques découvertes devient trop important il est difficile de réagir rapidement. Ainsi, de nombreux équipements et logiciels peuvent rester vulnérables longtemps après la découverte d'une vulnérabilité et encore plus longtemps si le correctif empêche la conformité du produit avec ses spécifications.

Pourtant, lorsqu'un client investit dans un équipement ou un service, celui-ci s'attend à ce que le produit acheté soit sûr et sécurisé. Que l'on soit un État [1], une entreprise [2], une célébrité [3] ou encore un simple inconnu [4], le besoin de

1. http://www.lemonde.fr/technologies/article/2011/03/07/l-elysee-et-les-affaires-etrangeres-egalement-touches-par-une-intrusion-informatique_1489767_651865.html
2. http://www.lemonde.fr/technologies/article/2011/11/11/la-plate-forme-steam-victime-d-une-intrusion-informatique_1602547_651865.html
3. http://www.lemonde.fr/pixels/article/2014/09/01/des-photos-piratees-de-jennifer-lawrence-et-plusieurs-autres-stars-nues-mises-en-ligne_4479584_4408996.html
4. http://www.lemonde.fr/technologies/article/2014/03/12/espionnage-la-nsa-utilise-des-logiciels-malveillants-a-une-echelle-industrielle_4381942_651865.html

garantir la confidentialité et l'intégrité de ses données sensibles est le même. Le fait que cette garantie soit dépendante de l'arrivée hypothétique d'un correctif de sécurité n'est donc pas acceptable.

La nécessité de déployer des mécanismes de protection qui peuvent garantir automatiquement l'application de propriétés de sécurité [3] afin de contrer dynamiquement l'exploitation de vulnérabilités [4], connues ou non, apparaît alors comme une évidence.

## 1.1 Apports de la thèse

Dans l'offre d'Alcatel-Lucent et celle d'autres fournisseurs, beaucoup de services et de middleware applicatifs sont basés sur Java. Ainsi les problèmes liés à Java affectent directement la sécurité et les coûts de maintenance des produits et services industriels [5] de ces entreprises. Or il n'existe pas de protection satisfaisante pour Java et encore moins de méthode pour calculer automatiquement les politiques de sécurité nécessaires. C'est pourquoi, cette thèse propose de définir, modéliser et implémenter une nouvelle approche de protection dédiée à Java.

Pour être déployée dans un contexte opérationnel, cette protection doit répondre à trois objectifs : efficacité, maintenabilité et performance. Si la littérature en vigueur incite à la mise en œuvre d'un contrôle d'accès obligatoire pour Java, la difficulté d'écriture des politiques de sécurité dans ce cadre est le frein principal à l'adoption de cette technologie. En effet, les approches obligatoires autorisent une granularité de contrôle très fine ce qui peut conduire à une politique composée de plusieurs dizaines de milliers de règles. En pratique, une telle complexité d'écriture n'est pas gérable manuellement et est difficilement maintenable. De plus, les développeurs/concepteurs industriels sont naturellement opposés à intégrer une solution qui ruinerait les performances de leur produit. Or une politique avec un tel niveau de complexité ne peut qu'avoir un impact important sur les performances. Sans oublier que la complexité de la politique n'est pas synonyme d'une bonne prise en compte des objectifs de sécurité. Ces trois difficultés majeures représentent les défis abordés par cette thèse.

Ainsi, pour être efficace, il est essentiel que cette protection satisfasse les objectifs de sécurité. En pratique, elle doit donc *a minima* contrer les malwares Java traditionnels, c'est à dire ceux exploitant des vulnérabilités de l'API Java ou de la JVM. Mais elle doit aussi pourvoir contrer des malwares jusqu'alors inconnus et qui visent soit des 0-day [6] Java, soit des défauts d'isolation dans les produits industriels pour lesquels il n'y a pas toujours de diffusion au public d'un

5. `http://arstechnica.com/security/2013/02/facebook-computers-compromised-by-zero-day-java-exploit/`

6. Il s'agit de vulnérabilités non-connues de l'éditeur.

rapport de vulnérabilité. Cela revient à développer un mécanisme de contrôle qui va au-delà du modèle de sécurité standard de Java ou même de celui d'un anti-virus.

Ensuite, pour être utilisable, il est essentiel que les efforts de configuration et de maintenance soient minimalistes. Parce que nous faisons le choix d'une approche par contrôle d'accès obligatoire, la politique de sécurité doit pouvoir se définir de façon fiable et sûre par rapport aux objectifs de sécurité. Cela implique que les politiques doivent se calculer dynamiquement à partir des requis de sécurité de l'application qui évoluent au cours de l'exécution de celle-ci. Cet objectif va au-delà des approches existantes qui consistent le plus souvent à traduire l'exécution observée d'un programme en politique de sécurité (principe de l'apprentissage) sans tenir compte des différents besoins de sécurité qui évoluent selon les états de l'application.

Enfin, pour être déployable dans un contexte opérationnel, la question des performances est critique. En effet, les produits industriels exigent des mécanismes de protection qui présentent un surcoût acceptable. Cela signifie que la protection et le calcul dynamique des politiques ne doivent pas impacter significativement les performances pour autoriser un passage à l'échelle d'un système industriel.

Au final, le choix de Java a pour origine un besoin industriel fort et un vide technologique avéré. Cette thèse propose :

— Un modèle de contrôle d'accès obligatoire pour systèmes à objets et notamment Java ;

— Un procédé de calcul dynamique des politiques de sécurité obligatoires généralisé et supportant les environnements Java ;

— Un moyen de réutiliser les politiques de sécurité existantes qui caractérisent les objectifs de sécurité comme, par exemple, les spécifications des besoins de sécurité Java ;

— La démonstration de l'efficacité pratique de l'approche via l'utilisation d'un logiciel professionnel de cyberattaque.

## 1.2 Plan du mémoire

Pour présenter ces quatre contributions, la thèse s'articule de la façon suivante :

1. Nous proposons un état de l'art en chapitre 2 qui illustre les vulnérabilités Java ainsi que le modèle standard de sécurité Java. Ensuite, nous introduisons les différents travaux de recherche visant à améliorer la sécurité de Java. Cela montre un manque en matière de protection obligatoire pour Java bien que la méthode *Java Authentication & Autorization Services (JAAS)* soit pertinente pour spécifier les besoins de sécurité. Cependant, l'absence d'un véritable moniteur de référence garantissant le respect des politiques fragilise l'approche JAAS.

2. Dans le chapitre 3, nous partons du constat précédent et montrons qu'il n'existe pas de modèle général satisfaisant pour contrôler les relations au sein des systèmes à objets. C'est pourquoi nous proposons un modèle général de systèmes à objets et montrons qu'il supporte les grandes classes de systèmes (langages non-objets, à classes, à prototypes, et à objets répartis). Ce modèle permet de typer dynamiquement les objets et ainsi leur attribuer un contexte de sécurité nécessaire pour exprimer/appliquer les règles de contrôle d'accès obligatoire. Cette modélisation permet aussi de couvrir un large ensemble de contextes tout en offrant une granularité de contrôle très fine. Nous montrerons que ce modèle couvre effectivement les systèmes à classes, à prototypes et à objets répartis. Ce chapitre se poursuit par une définition exhaustive des relations entre objets qu'il est possible de contrôler. Nous distinguons ainsi les notions de références, d'interactions et trois types de flux (d'informations, de données et d'activités). Nous proposons un modèle de logique général qui utilise ces différents types de relations. L'idée directrice est celle d'une spécification des besoins de sécurité sous la forme d'un automate qui utilise ces relations.

3. Le chapitre 4 continue en montrant qu'il est possible de projeter le modèle JAAS sous la forme de nos automates afin de réutiliser les objectifs de sécurité définis dans Java. Ainsi, nous rendons explicites pour notre contrôle d'accès obligatoire les besoins de sécurité spécifiés au moyen de JAAS. Aussi, nous commençons par rappeler les limitations de JAAS en termes de garantie et proposons les algorithmes pour calculer l'automate qui va contrôler l'exécution de l'application. On illustre ainsi que l'utilisation d'automates dépasse largement la notion classique de contrôle d'accès puisque l'on peut, par ce biais, garantir dynamiquement différents objectifs d'exécutions de l'application à des relations complexes entre objets du système. Par la suite, nous reprenons le modèle d'attaque défini dans notre état de l'art et montrons que nous pouvons effectivement les prévenir au moyen de nos automates. Enfin, nous montrons que notre modèle définit au chapitre 3 se projette bien sur un modèle de contrôle d'accès obligatoire de type *Domain & Type Enforcement (DTE)*. Ainsi, nous passons d'un modèle de contrôle abstrait à un modèle de contrôle concret qui peut être piloté par nos automates. Afin d'illustrer la génération automatique de politiques, nous établissons une méthode de traduction dynamique des politiques JAAS en politique DTE.

4. Le chapitre 5 présente le prototype de recherche que nous avons déve-

loppé au cours de cette thèse. Pour cela, nous commençons par présenter les avantages et inconvénients de différentes techniques d'instrumentation d'une machine virtuelle Java (JVM) dont le développement ou la modification d'une JVM, l'utilisation d'un agent JVMTI et l'injection de bytecodes Java. Nous en concluons que la première approche est la mieux adaptée pour intégrer un véritable moniteur de référence dans la JVM bien que celle-ci nécessite un effort de développement conséquent. Les autres approches sont quant à elles plus simples à mettre en œuvre mais au prix d'une moins bonne intégration au sein de la JVM. Nous continuons ensuite en présentant les principaux composants logiciels de notre moteur de décisions qui inclut une logique multi-niveaux (MLS), une logique DTE et notre approche par automates pour traduire les politiques JAAS en politique DTE. Nous continuons ce chapitre par le contrôle d'un malware Java tout en détaillant, à chaque étape de son exécution, les calculs et actions de notre moteur de décisions. Enfin, nous terminons cette thèse en présentant les résultats obtenus avec plusieurs autres malware Java.

Pour faciliter la lecture de cette thèse, le lecteur trouvera en page 209 un index de nos hypothèses, notations et suggestions.

Chapitre 2

# État de l'art

## Sommaire



*La vulnérabilité croissante des systèmes est une autre classe de problèmes non résolus. Les systèmes deviennent plus gros, plus rapides, et cela laisse de plus en plus de place pour que quelque chose tourne mal. Nous pourrions essayer de contenir les incidents, ce qui n'est pas une tâche triviale car par leur structure ces systèmes peuvent propager le moindre bit de confusion à la vitesse de la lumière.* [· · ·] *Nous sommes maintenant confrontés à ce problème à large échelle et il ne s'agit plus d'un problème de gestion mais d'un défi scientifique.*

Edsger W. Dijkstra [5]

## 2.1 Historique des vulnérabilités Java

Java est une technologie très répandue que l'on retrouve dans de nombreux équipements tels que les machines de bureau, les serveurs et les mobiles. Mais on en trouve aussi dans les équipements multimédia, les systèmes avioniques [1] et les cartes à puce pour ne citer que eux. Oracle, l'éditeur principal de Java, avance même le chiffre de 5 milliards d'équipements sur son site officiel [2].

Première conséquence de cette popularité : tous les équipements, produits et services basés sur Java sont automatiquement affectés par les problèmes de sécurité liés à cette technologie. Seconde conséquence : l'existence d'une vulnérabilité Java exploitable présente un intérêt majeur pour un attaquant désireux de compromettre un maximum de machines.

C'est pour ces raisons que l'on peut observer des vagues de cyber-attaques visant explicitement des vulnérabilités Java. L'auteur de [6] propose un historique sur la découverte de ces vulnérabilités dont nous nous permettons de reprendre les graphiques dans les figures 2.1 et 2.2. On y observe tout d'abord une accélération dans le nombre de vulnérabilités Java découvertes (cf. figure 2.1) qui semble se poursuivre avec les chiffres disponibles pour le premier trimestre 2013 (cf. figure 2.2). Cela montre bien l'intérêt grandissant des cyber-attaquants pour les problèmes de sécurité liés à Java depuis la sortie de la première version en 1996.

Ces statistiques mettent également en lumière un pic de découvertes de vulnérabilités en 2009 puis un autre en 2012/2013. Ces pics coïncident avec les audits de sécurité qui ont eu lieu après la découverte respective des vulnérabilités CVE-2008-5353 [3] et CVE-2012-4681 [4]. Ces deux vulnérabilités sont connues pour avoir engendré des vagues de cyber-attaques massives du fait de leur portabilité et de leur taux d'exploitabilité de 100%.

Par ailleurs, au regard de l'intensité des cyber-attaques visant Java en 2012 et des difficultés d'Oracle à proposer des correctifs de sécurité pérennes, le département de la sécurité intérieure des États-Unis avait fortement recommandé d'abandonner la technologie Java [5]. Malheureusement cela nécessiterait de se débarrasser des téléviseurs, cartes bleues, smartphones, serveurs d'applications, systèmes avioniques et d'une bonne partie des infrastructures informatiques actuelles au profit d'une technologie concurrente pas nécessairement plus efficace en termes de sécurité.

1. http://www.jcraft.com/news/06.18.2009pr.html
2. https://www.java.com/fr/about/
3. http://www.rapid7.com/db/modules/exploit/multi/browser/java_calendar_deserialize
4. http://www.rapid7.com/db/modules/exploit/multi/browser/java_jre17_exec
5. http://www.kb.cert.org/vuls/id/625617

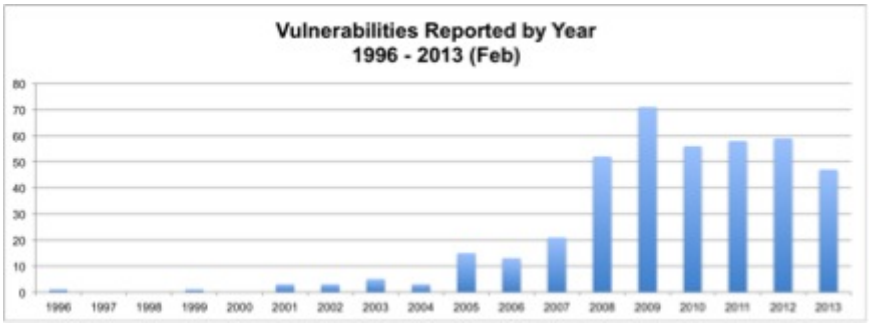



FIGURE 2.1 – Vulnérabilités Java découvertes entre 1996 et 2013


FIGURE 2.2 – Vulnérabilités Java découvertes entre février 2009 et février 2013

C'est pourquoi il est nécessaire de proposer un mécanisme de sécurité intégré à la machine virtuelle Java, complétant les mécanismes Java existants et supportant les applications Java existantes sans que cela nécessite de les recompiler.

## 2.2 Modèle d'attaques Java

Étant donné que l'objectif est au minimum de prévenir les vulnérabilités Java connues, nous commençons par analyser les types de vulnérabilité identifiés dans Java. Nous présentons les raisons pour lesquelles ces types d'attaque restent d'actualité.

### 2.2.1 Corruption de mémoire

Par nature, certaines vulnérabilités reposent sur un bug informatique qui entraîne un problème de sécurité. De nombreux bugs informatiques consistent en une corruption de la mémoire c'est-à-dire une mauvaise modification de la mémoire (ex : mauvaise initialisation de pointeurs, débordement de tampons, etc.). L'espace mémoire des applications est protégé de ces corruptions par les vérifications que fait la machine virtuelle. Cependant l'hyperviseur Java qui contrôle les objets Java n'est pas protégé de ces corruptions.

Statistiquement, la probabilité d'apparition de bugs et in-fine de vulnérabilités dans une implémentation de l'hyperviseur Java (JRE) n'est pas nulle. C'est pourquoi on découvre parfois des vulnérabilités liées à Java mais que l'on pensait spécifiques aux applications natives telle que la CVE-2010-0842 [6].

En pratique ce type d'attaque au niveau de l'hyperviseur Java est relativement rare car sa taille est petite en comparaison avec la taille d'un noyau de système d'exploitation traditionnel tel que Windows ou Unix. Étant donné que les problèmes au niveau de l'hyperviseur sont bien maîtrisés nous considérons que ce type d'attaque sort du cadre de notre étude. Cependant, nous nous intéressons à prévenir les corruptions mémoire qui existent au niveau de l'espace mémoire Java. En effet, même si la machine virtuelle contrôle bien l'espace Java, il reste une probabilité importante de corruption de l'espace Java violant l'intégrité de celui-ci. Notre étude visant à améliorer l'intégrité de l'espace mémoire Java, nous allons citer des types d'attaques sur celui-ci.

6. `http://www.rapid7.com/db/modules/exploit/windows/browser/java_mixer_sequencer`

### 2.2.2 Confusion de type

Java est un langage fortement typé et l'utilisation d'une machine virtuelle pour exécuter des programmes Java (JVM) apporte la possibilité de réaliser des vérifications à la volée (cf. section 2.3.1). Parmi ces vérifications, il existe un contrôle strict sur le forçage de types entre deux objets Java. L'attaque par confusion de type consiste alors à exploiter un bug dans l'implémentation de cette vérification ou une erreur matérielle [7] afin de contourner les limitations imposées sur la visibilité des membres d'une classe privilégiée de l'application ou de l'API Java [8].

L'illustration la plus simple d'une telle attaque est l'usurpation (spoofing) d'une classe Java système. Si l'attaquant usurpe le type d'une instance de cette classe système en le forçant en un type dont tous les membres sont de visibilité publique, alors cet attaquant pourra accéder sans restrictions à tous les membres de cet objet système. Par exemple, les auteurs de [9] réalisent une analyse de la vulnérabilité CVE-2012-0507 dont l'exploitation repose sur le principe de confusion de type.

On peut assimiler ce type d'attaque à une corruption mémoire de l'espace Java. Nous en concluons que contrôler plus finement l'espace mémoire Java, c'est à dire là où résident les objets Java, devrait permettre d'empêcher ce type de vulnérabilité.

En pratique ce type d'attaque est difficile à contrer car intimement lié à une implémentation de la JVM. Toutefois, les privilèges de la classe usurpatrice peuvent tout de même être limités en contrôlant finement ses interactions avec les autres objets Java.

### 2.2.3 Défaut de contrôle d'accès

Le langage Java propose des notions de protection (public/private/protected/-default) pour les objets Java qui s'appliquent aux méthodes et aux champs. Ces attributs de sécurité lorsqu'ils sont positionnés sur les objets participent à la garantie de l'intégrité et de la confidentialité des données et fonctionnalités sensibles d'une application.

Pour contrôler finement l'accès à ces éléments, le développeur se doit alors d'appeler des fonctions de contrôle d'accès vérifiant les privilèges de l'appelant vis à vis de l'objet appelé et de ses méthodes via l'API *Java Authentication and Autorisation Services* (cf. section 2.3.4). Le développeur doit donc écrire une partie des fonctions de contrôle d'accès.

En pratique, même s'il est du ressort de l'administrateur de la machine de définir les objectifs de sécurité, étant donné que le développeur participe à la garantie de ces objectifs, des développeurs défaillants ou malicieux peuvent compromettre la confidentialité et/ou l'intégrité de l'environnement Java voire éventuellement prendre le contrôle du système d'exploitation hôte. Selon [10], il s'agit ici de la principale origine des problèmes de contrôle d'accès dans Java.

L'objectif de notre étude est donc de prévenir les développeurs défaillants ou malicieux en garantissant les objectifs définis par l'administrateur indépendamment des contrôles fait par programmation. Nous détaillerons dans la suite de cet état de l'art le principe de JAAS avec les liens entre le rôle de l'administrateur et du programmeur puisque celui-ci est au cœur de la majorité des attaques sur Java.

### 2.2.4 Abus de l'inspection de pile

Les auteurs de [11] expliquent que l'obtention d'une référence sur un objet privilégié de la JVM n'est pas problématique en soit. En effet, il peut être légitime qu'un objet non privilégié puisse appeler un objet privilégié. Mais les vérifications de privilèges réalisées par l'inspection de pile se limitent dans certains cas à l'objet privilégié, ce qui fait que l'objet appelant non privilégié peut ainsi élever ses privilèges. Ce comportement est confirmé par la documentation officielle de Oracle [12]. Ainsi ces mêmes auteurs détaillent plusieurs méthodes génériques pour abuser de cette faiblesse du contrôle d'accès de Java en appelant des méthodes privilégiées de l'API Java.

Par exemple, l'API de réflexion permet d'accéder à des méthodes privilégiées. Ceci est confirmé par [10] qui souligne l'aspect sensible de cette API système car capable de contourner les restrictions d'accès via les objets privilégiés de la JVM.

En pratique, un mécanisme d'inspection de pile plus élaboré que celui de la JVM existante devrait être capable de contrôler ces élévations de privilèges.

## 2.3 Modèle de sécurité standard de Java

Conceptuellement il n'y a pas de différence entre une machine virtuelle Java (JVM) et une machine virtuelle Virtualbox, VMWare ou autre. Ce sont toutes des implémentations logicielles d'un processeur abstrait, d'un gestionnaire de mémoire et d'un contrôleur d'entrées/sorties. Ainsi la JVM est constituée d'un

hyperviseur, de classes Java systèmes et de classes invitées. La différence principale avec un système d'exploitation invité, via Virtualbox par exemple, est qu'un invité de l'hyperviseur Java est constitué de classes Java et non pas de binaires.

La licence attachée à Java autorise n'importe quel éditeur à proposer son implémentation de JVM à condition de respecter les spécifications techniques de la machine virtuelle [13] et du langage [14]. Ainsi, tout programme Java est compatible avec la machine virtuelle de Sun/Oracle (Hotspot) mais aussi avec celle d'éditeurs tiers comme Avian [7], Apple [8] ou encore IBM [9]. L'hyperviseur Java de Google pour Android (Dalvik [10] [15]) ne respecte pas entièrement les spécifications de la machine virtuelle [13] mais supporte bien le langage Java [14].

Cette section propose une analyse des différents mécanismes de sécurité présents en standard dans les machines virtuelles Java. Une analyse plus détaillée du fonctionnement interne d'une JVM est disponible dans [16].

### 2.3.1 Sécurité du langage

Le langage Java est souvent considéré comme sûr par la communauté de développeurs car il n'y a pas, par exemple, de gestion explicite de la mémoire ni de manipulation directe des pointeurs contrairement aux langages C et C++. De plus, l'utilisation d'une machine virtuelle permet de vérifier à la volée certaines opérations dangereuses et détecter par exemple les débordements de tampon ou le formatage de chaînes de caractères non contrôlées (cf. figure 2.3).

```
1 import java.io.*;
2
3 public class FormatString {
4   public static void main(String[] argv) {
5     System.out.printf(argv[0]); // <-- argv[0] == "%s"
6   }
7 }
```

FIGURE 2.3 – Exemple d'une vulnérabilité de type Format String non exploitable dans les environnements Java.

Une exception de type *MissingFormatArgumentException* est levée si *%S* est passée en argument. Il s'agit ici d'une exception silencieuse qui n'est pas gérée par le programme.

Pourtant, un langage sûr n'implique pas nécessairement du code exempt de bugs ou de vulnérabilités. Ainsi, l'analyse réalisée par [6] montre une accélération

7. http://oss.readytalk.com/avian/
8. http://support.apple.com/fr_FR/downloads/#java
9. http://www.ibm.com/developerworks/java/jdk/
10. https://source.android.com/devices/tech/dalvik/index.html

dans la découverte de vulnérabilités liées à Java. Cette tendance est confirmée en regardant le rythme des contributions dans les bases d'exploits publiques (Metasploit [11], 1337day [12], etc.).

C'est pourquoi, l'Agence Nationale pour la Sécurité des Systèmes d'Information (ANSSI) propose un ensemble de recommandations liées à l'utilisation et au développement d'applications Java [10]. Cependant, l'usage stricte de ces recommandations semble difficile en pratique. Il est donc préférable d'avoir des méthodes de protection en profondeur au sein de la JVM afin de pallier les erreurs de programmation.

### 2.3.2 Chargeurs de classes et isolation mémoire

Afin de compartimenter l'espace d'adressage de la JVM, différents chargeurs de classes (classloader [13, 16]) sont utilisés. En pratique l'espace d'adressage est ainsi subdivisé en différentes zones mémoire qui sont isolées entre elles. Cependant des interactions existent entre ces différents espaces mémoire. Par exemple, un objet de la zone application peut appeler un objet de la zone système.

Un chargeur de classe (classloader) importe à la volée les classes Java en vérifiant au-préalable le format du *.class* importé, le typage, les références à d'autres classes. En pratique, un classloader garantit l'unicité des classes dans l'espace de nom dont il a la responsabilité [17]. En standard, la JVM implémente trois classloaders [13] qui peuvent être étendus en fonction des besoins :

- **BootstrapClassloader** pour charger les classes nécessaires au démarrage et au fonctionnement de la JVM (les classes noyau) ;
- **SystemClassloader** pour charger les extensions de l'API Java et les classes systèmes ;
- **ApplicationClassloader** pour charger le programme Java de l'utilisateur.

L'isolation mémoire offerte par le classloader est au cœur du modèle de sécurité de Java. La section 2.3.3 détaille ce lien entre le classloader Java et les mécanismes d'autorisation de Java faisant parties de JAAS. Cependant, des attaques existent sur le mécanisme de chargement qui passent généralement par des défauts de contrôle d'accès et d'inspection de pile tels que décrits dans la section précédente.

11. `http://www.metasploit.com/`
12. `http://www.1337day.com/`

### 2.3.3 Java Authentification & Authorization Services

La *Java Authentication & Authorization Services* (JAAS) est une API de sécurité en standard depuis Java 1.1, qui implémente des primitives d'authentification et de contrôle d'accès [18]. Seule la partie *authorization* de JAAS nous intéresse car, comme nous l'avons vu, les faiblesses de Java reposent essentiellement sur les fragilités du contrôle d'accès.

JAAS s'appuie sur un contrôle d'accès traditionnel puisqu'il permet à un administrateur d'écrire facilement des règles du type Sujet $\xrightarrow{\{\text{Permissions}\}}$ Objet . Cependant, la mise en œuvre du contrôle nécessite une intervention des programmeurs sur la base d'une spécification. C'est cette intervention qui constitue une des sources d'erreurs et fragilise la garantie du respect des politiques.

Avec JAAS, l'administrateur définit des capacités (C) pour les différents sujets (S), ce qui correspond à une partie de la règle de contrôle soit S $\longrightarrow$ C . Par ailleurs, le programmeur a la charge de réutiliser des spécifications, définissant à quoi correspondent les capacités soit C :$\{P_1 \longrightarrow O_1, P_2 \longrightarrow O_2, \ldots\}$, afin de coder le contrôle d'accès correspondant au sein des objets ($O_1$, $O_2$, ...).

Ainsi, par la combinaison du travail de l'administrateur et de l'implantation par le programmeur du contrôle d'accès nécessaire aux capacités (cf. figure 2.4), JAAS permet de définir et de garantir des règles de contrôle d'accès élaborées S $\xrightarrow{\{P_1\}} O_1$ , S $\xrightarrow{\{P_2\}} O_2$ , ...

$$\overbrace{(\text{Sujet JAAS}) \underbrace{\xrightarrow{\{\text{Capacité}\}}}_{\text{Développeur}} (\text{Objet Java})}^{Administrateur}$$

FIGURE 2.4 – Répartition des responsabilités dans la définition des politiques Java

Les politiques Java reposent sur des capacités. Le développeur traduit les capacités définies dans les spécifications en points de contrôle dans le corps des méthodes Java. L'administrateur utilise ces capacités pour autoriser des sujets JAAS à accéder à des ressources ou des fonctionnalités de l'application.

On peut déjà faire trois remarques.

Premièrement, le travail de l'administrateur est facilité puisque celui-ci manipule seulement des capacités et n'a donc pas à connaître le détail des règles de contrôle d'accès associées aux capacités. Il bénéficie ainsi d'une factorisation des règles S $\xrightarrow{\{\text{P}\}}$ O sous la forme S $\longrightarrow$ C qui se fait au détriment d'un manque de finesse et de visibilité.

Deuxièmement, le détail du contrôle d'accès associé à une capacité peut évoluer facilement puisque les politiques des administrateurs ignorent ce détail. Ainsi, l'administrateur n'aura pas à changer ses règles si la spécification évolue. Cependant, les évolutions doivent être connues de l'administrateur afin qu'il sache si ses objectifs de sécurité restent satisfaits. On voit que la facilité d'évolution n'est pas favorable à la visibilité du contrôle requis.

Troisièmement, la satisfaction des règles de contrôle d'accès est du ressort du programmeur. En effet, c'est lui qui a la charge de placer les points de contrôle dans les différents objets concernés par la capacité. Par exemple, il devra placer un point de contrôle dans la méthode $P_1$ de l'objet $O_1$ et un point de contrôle dans la méthode $P_2$ de l'objet $O_2$. Ainsi, la garantie est faible car dépendante du programmeur.

### Administration des besoins de protection

L'administrateur de la machine hôte peut définir les capacités des sujets. Les sujets sont définis au moyen de différentes informations (*signedBy*, *codeBase* et *principal*). Une capacité est définie au moyen d'un attribut *permission*. La figure 2.5 donne la syntaxe correspondante. Ainsi, l'administrateur autorise (*grant*) un signataire, une localisation du code (*codeBase*), une identité (*principal*) ou les trois à accéder à une fonctionnalité c'est à dire lui donne une capacité (*permission*). JAAS applique le principe du moindre privilège [19] c'est-à-dire que tout ce qui n'est pas explicitement autorisé est interdit. L'administrateur n'a donc qu'à autoriser un ensemble de capacités, les capacités restantes étant naturellement interdites selon ce principe. On voit que JAAS repose sur une bonne pratique de sécurité.

```
1  grant signedBy "signer_names", codeBase "URL",
2    principal principal_class_name "principal_name",
3    principal principal_class_name "principal_name",
4    ... {
5
6    permission permission_class_name "target_name", "action",
7      signedBy "signer_names";
8    permission permission_class_name "target_name", "action",
9      signedBy "signer_names";
10   ...
11 };
```

FIGURE 2.5 – Syntaxe des fichiers de politique JAAS

Le rôle du classloader est essentiel puisque c'est lui qui a la responsabilité de calculer, au chargement des classes, les sujets et les permissions associés à chaque classe. Ainsi, le contexte de sécurité JAAS de la classe de même que ses privilèges sont stockés dans un objet de type `java.security.ProtectionDomain` [13] associé à cette classe [14] (cf. figure 2.6).

13. `http://docs.oracle.com/javase/7/docs/api/java/security/ProtectionDomain.html`
14. `http://docs.oracle.com/javase/7/docs/api/java/lang/Class.html#getProtectionDomain()`

Dans la terminologie Java le sujet est appelé *domaine de protection* et comprend le signataire du .jar, l'URL du .jar/.class et l'identité des principaux Java. Certaines classes ne sont pas soumises au contrôle parce qu'il s'agit de classes privilégiées telles que les classes système ou noyau. C'est la raison pour laquelle elles sont chargées par des classloaders dédiés qui bien que n'effectuant pas de contrôles garantissent une isolation mémoire de ces classes privilégiées.

```
1  public class ProtectionDomain {
2
3    /* CodeSource */
4    private CodeSource codesource ;
5
6    /* ClassLoader the protection domain was consed from */
7    private ClassLoader classloader;
8
9    /* Principals running-as within this protection domain */
10   private Principal[] principals;
11
12   /* the rights this protection domain is granted */
13   private PermissionCollection permissions;
14
15   /* if the permissions object has AllPermission */
16   private boolean hasAllPerm = false;
17
18   /* the PermissionCollection is static (pre 1.4 constructor)
19     or dynamic (via a policy refresh) */
20   private boolean staticPermissions;
21 }
```

FIGURE 2.6 – Source de la classe `java.security.ProtectionDomain`

Les champs de l'objet représentant le domaine de protection de la classe conformément à la syntaxe des politiques Java (cf. figure 2.5). De plus, on note la présence d'une référence sur le classloader afin d'indiquer dans quel espace d'adressage se situe la classe (kernel, système, application, ...).

En pratique, un domaine de protection Java contient les informations de haut niveau qui abstraient les mécanismes internes tels que la signature de code, l'authentification d'utilisateurs et les capacités. Cette abstraction facilite la définition des politiques Java. Cependant, l'utilisation et la mise en œuvre de ces abstractions ne sont pas immédiates ni complètement sûres. Notamment, nous l'avons vu, la satisfaction des capacités est à la charge du programmeur des classes correspondant aux objets.

### 2.3.4 Définition et satisfaction des capacités

Comme expliqué précédemment, une capacité abstrait un ensemble de privilèges sur différents objets. Par exemple, les spécifications de sécurité pour Java 1.7 [20] définissent la capacité `java.lang.RuntimePermission "loadLibrary"` comme permettant d'appeler les méthodes `load()` et `loadLibrary()` des classes `java.lang.Runtime` et `java.lang.System`. Le développeur de ces deux classes doit alors coder les méthodes `load()` et `loadLibrary()` en vérifiant que le sujet du code appelant possède bien la capacité nécessaire ; ici la capacité `"loadLibrary"`, par exemple.

Ces spécifications montrent également que l'accès à une méthode peut être autorisé par différentes capacités qui sont sélectionnées au moyen d'une logique associée à cette méthode. Par exemple, la méthode `send()` de la classe `java.net.DatagramSocket` est associée à une logique qui sélectionne la capacité `java.net.SocketPermission "accept,connect"` si l'adresse d'envoi est de type multicast ou la capacité `java.net.SocketPermission "connect"` pour une adresse unicast. Ainsi, la capacité requise pour la méthode `send()` varie.

Il revient donc aux spécifications de définir la logique d'association entre les capacités et les objets concernés. Cependant, le point intéressant est que cette logique est avancée puisqu'elle peut utiliser l'expressivité du langage Java pour affiner encore. Par contre, il est du ressort du développeur de mettre en place cette logique au sein des méthodes des objets concernés. Ce qui laisse la place à des erreurs de codage. Ainsi l'extrait de code du contrôle d'accès à la méthode `load()` de `java.lang.Runtime` montre que le programmeur doit faire un appel explicite à JAAS via l'objet `SecurityManager` (cf. figure 2.7). Nous présentons le rôle du `SecurityManager`, c'est à dire du contrôleur JAAS, dans la section suivante.

L'approche est extensible car elle permet à un développeur de définir par programmation une nouvelle capacité. Pour cela, le programmeur définit le nom de la capacité, une liste de permissions et les ressources concernées. Par exemple, le développeur crée une classe `java.io.FilePermission` définissant le nom de la capacité, la liste de permissions (`read`, `write`, ...) qui sera contrôlée par cette classe ainsi qu'un type de ressource ici un nom de fichier. Une capacité peut être directement une fonctionnalité de la JVM ou de l'application et dans ce cas la capacité se réduit à un nom.

```
1  public void load(String filename) {
2    SecurityManager security = System.getSecurityManager();
3    if (security != null) {
4      security.checkPermission(new RuntimePermission("loadLibrary."+filename));
5    }
6
7    ...
8
9    ClassLoader.loadLibrary(fromClass, filename, true);
10 }
```

FIGURE 2.7 – Code de la méthode `load` protégée par la capacité `"loadLibrary"`.

### Security Manager

Comme présenté ci-dessus, les développeurs posent des points de contrôle dans les objets concernés par une capacité en faisant un appel à l'objet de type `java.lang.SecurityManager`. Celui-ci est un proxy de la classe `java.security.AccessController` qui vérifie les privilèges pour chaque domaine de protection présent dans la pile d'appels.

```
1 SecurityManager security = System.getSecurityManager();
2 if (security != null) {
3   security.checkXXX(argument, . . . );
4 }
```

FIGURE 2.8 – Code de référence pour invoquer JAAS

Cette portion de code est généralement placée en entête de la méthode à protéger, de manière à constituer les premiers bytecodes exécutés au moment de l'appel. Le développeur précise la permission Java que le code appelant doit posséder pour continuer l'exécution. Si le code appelant ne possède pas le privilège demandé par le développeur alors une exception de type `java.lang.SecurityException` est levée.

En plaçant le contrôle d'accès hors de portée de l'utilisateur de l'application, JAAS suit clairement une approche obligatoire puisque l'utilisateur final n'a pas la possibilité de définir ou modifier la politique. Par contre, la méthode mise en œuvre par JAAS pour garantir cette politique ne respecte pas entièrement les critères d'un moniteur de référence de Anderson [21] car le moniteur de référence de JAAS est désactivable par programmation [15]. C'est pourquoi les développeurs doivent vérifier si un `SecurityManager` est bien présent (cf. figure 2.8). En conséquence, JAAS ne propose pas de réel moniteur de référence ce qui ne permet pas d'en faire une approche obligatoire satisfaisante.

### Inspection de pile

Comme décrit précédemment, le `SecurityManager` de JAAS utilise la classe `java.security.AccessController` qui réalise une inspection de la pile d'exécution (Stack Based Access Control - SBAC). Un tel mécanisme s'appuie sur une analyse en profondeur du contenu de la pile d'exécution du thread au moment du contrôle d'accès, d'où le nom de cette approche. Apparue initialement avec Netscape, les machines virtuelles comme Java [22, 23] ou les environnements d'exécution comme CLR [24] utilisent l'inspection de pile pour empêcher les élévations de privilèges non contrôlées.

En pratique, la classe `java.security.AccessController` construit une copie de la pile dans `java.security.AccessControlContext` [16] qui contient la liste des domaines de protection de la pile d'appels. Ainsi, la classe vérifie que tous les contextes de la pile ont bien les privilèges demandés par l'objet en haut de la pile. Ainsi, on évite les élévations de privilèges d'une méthode non-privilégiée qui appelle une méthode privilégiée.

Cependant en pratique, il est nécessaire d'autoriser certaines élévations de privilèges pour des raisons fonctionnelles. Par exemple, un objet non privilégié qui nécessite l'utilisation d'une classe système. JAAS propose notamment la primitive

15. `http://docs.oracle.com/javase/7/docs/api/java/lang/System.html#setSecurityManager(java.lang.SecurityManager)`
16. `http://docs.oracle.com/javase/7/docs/api/java/security/AccessControlContext.html`

`doPrivileged()` pour autoriser une élévation de privilège. Toutes les actions passées en paramètre de cette méthode [17] seront exécutées avec les privilèges de l'objet appelant immédiat. En d'autres termes, cela ordonne à JAAS ne pas inspecter la pile d'exécution au delà de l'objet ayant appelé `doPrivileged()`. Comme l'indique la documentation officielle [12], cette fonctionnalité de JAAS induit un risque d'élévation de privilèges non contrôlé puisqu'elle désactive tous les contrôles à un certain niveau de la pile. Dans l'exemple simplifié qui suit on voit que le contrôle s'arrête au niveau de $o_1$ ce qui autorise une élévation non contrôlée et potentiellement malveillante pour l'objet $o_0$ qui obtient le privilège $P_3$ sur l'objet $o_3$.

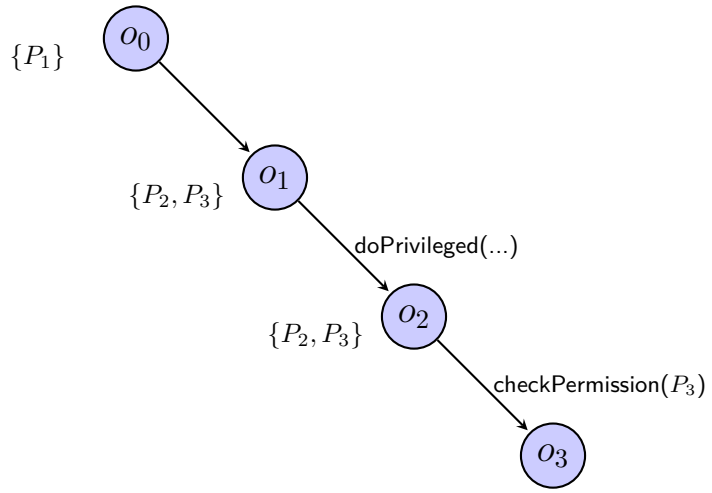



FIGURE 2.9 – Illustration du fonctionnement de `doPrivileged()`.

$o_0$ appelle $o_1$, puis $o_1$ appelle $o_2$ qui appelle $o_3$.

En conséquence ; pour certaines interfaces de programmation comme l'API de réflexion Java [11], les risques induits sont difficilement maîtrisables comme le rappel l'ANSSI [10] ou le CERT [25].

Bien qu'il existe différentes limitations sur l'inspection de pile dont nous parlerons ultérieurement, une des fragilités de l'approche JAAS nous semble être la possibilité de la désactiver à partir d'un certain niveau de la pile. Cela peut être pertinent si le comportement du programme est maîtrisé et présente un certain niveau de sécurité mais cette désactivation de l'inspection de pile reste risquée.

## 2.4 Amélioration de la sécurité Java

Il existe plusieurs façons de contrôler les flux d'information à l'intérieur d'une application Java dont certaines sont des transpositions à Java d'approches pré-

17. `http://docs.oracle.com/javase/7/docs/api/java/security/PrivilegedAction.html`

sentes dans les systèmes d'exploitation étant donné que la JVM a une architecture proche. Cette section se propose de passer en revue les approches relatives à l'amélioration du modèle de sécurité de Java.

### 2.4.1 Inspection de pile augmentée

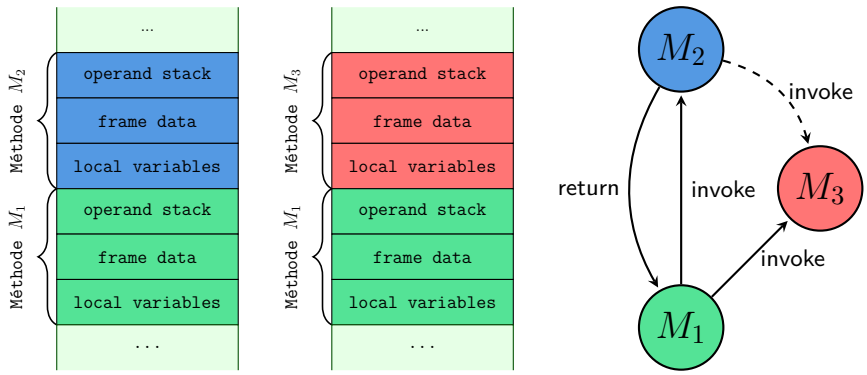
L'inspection de la pile d'exécution est une méthode de contrôle d'accès qui détermine si un fil d'exécution (thread) peut appeler une méthode privilégiée en tenant compte des appels précédents. L'implémentation d'un tel mécanisme s'appuie principalement sur l'analyse des privilèges des éléments contenus dans la pile d'exécution du thread, d'où son nom.

La difficulté de l'approche est d'être en mesure de capturer les appels aux méthodes à protéger. Dans le cas de Java, le développeur a la responsabilité d'instrumenter son programme avec du code Java en charge de déclencher l'inspection de la pile (cf. section 2.3.4). L'inconvénient majeur de s'en remettre à l'expertise du développeur est que cela enlève à l'expert en sécurité toute flexibilité sur le choix des méthodes à contrôler et sur les privilèges à vérifier. Ainsi, la mise en oeuvre du contrôle n'est pas sous la responsabilité d'un réel moniteur de références.

D'ailleurs, les auteurs de [26] expliquent que l'inspection de pile ne tient pas compte des méthodes dont le contexte d'exécution a disparu de la pile d'appels (cf. figure 2.10) ce qui peut potentiellement contourner la politique de sécurité. La solution des auteurs est de garder en mémoire tout l'historique d'exécution d'un thread afin de traiter l'aspect transitif des propriétés de sécurité. Mais il ne peut y avoir d'utilisation réaliste de cette solution car l'historique peut grandir jusqu'à une saturation de la machine virtuelle Java. De plus, cette approche a pour inconvénient que l'appel à une méthode moins privilégiée empêchera tout appel ultérieur à une méthode privilégiée. Cela ne répond donc pas à la nécessité d'autoriser certaines élévations de privilèges.

Dans [27], les auteurs considèrent les flux d'information entre méthodes plutôt que les privilèges du code précédemment exécuté. Avec une approche par preuve formelle et de l'analyse dynamique de code, ils montrent qu'ils sont en mesure de garantir des propriétés d'intégrité pour chaque fil d'exécution. Cependant, la complexité algorithmique de leur solution ne semble pas satisfaire à une mise en œuvre réaliste.

Au final, on constate que l'inspection de pile est certes en mesure de contrôler quelques élévations de privilèges mais pas toutes. Certaines élévations sont même très difficiles à contrôler comme celles impliquant plusieurs threads.




(a) $M_1$ appelle $M_2$ (b) $M_1$ appelle $M_3$ (c) Flux d'informations

FIGURE 2.10 – Exemple d'un flux non-détectable par inspection de pile

Si $M_1$ appelle $M_2$ (a), puis $M_1$ appelle $M_3$ (b) alors la stack frame de $M_3$ écrase celle de $M_2$. Il s'agit ici d'un flux indirect entre $M_2$ et $M_3$ (c) qui n'apparaît pas dans la pile d'exécution du thread.

### 2.4.2 Analyse de programmes

Le but de l'analyse de programmes est de pouvoir prédire les flux d'informations non-sûrs soit à partir des sources du programme soit durant son exécution. De façon générale, il existe une intense activité de recherche autour de l'analyse de programmes où l'auteur de [28] compte par exemple plus d'une centaine de publications sur le sujet. La littérature en vigueur s'appuie essentiellement sur les flux de données entre composants du programme. Un état de l'art sur les méthodes d'analyse des bytecodes Java est d'ailleurs proposé par [29].

Cependant le théorème de Rice [30] démontre que la garantie de propriétés de sécurité à partir de l'analyse des sources d'un programme est un problème indécidable. En d'autres termes, il n'existe pas d'algorithmes en mesure de certifier l'application d'une propriété de sécurité en un temps et un espace fini avec cette méthode.

```
1 // RICE theorem illustration
2 int main(int argc, char * argv[]) {
3   int x = 1;
4   printf("(%d %d %d)\n", x++, x, ++x);
5 }
```

FIGURE 2.11 – Illustration du théorème de H.G. Rice

L'incrémentation de la variable entière `x` se fait lors de l'appel à la fonction `printf`. En pratique cette fonction affichera soit `(1 2 3)` soit `(2 2 2)` car l'ordre d'évaluation de ses arguments est à la discrétion du compilateur [31].

Ainsi, ce type d'approche se limite souvent à la vérification de quelques flux comme ceux sortant du programme. Les effets de bord sont multiples (cf. listing 2.11) et il est difficile en pratique de garantir des propriétés de sécurité à

l'échelle d'un programme complet. C'est pourquoi la JVM se limite à garantir la conformité des `.class` par rapport aux spécifications techniques de Java.

### 2.4.3 Coloration dynamique

L'idée de la coloration dynamique est de suivre la propagation d'une donnée au travers du programme et ainsi surveiller les flux de l'application [32, 33]. Le principe est d'attribuer une étiquette, appelée couleur, à chaque variable du programme. Puis, pendant l'exécution du programme, chaque entité qui réalise une opération de lecture sur cette variable est contaminée par la couleur qui lui est associée. Par transitivité, le programme est ainsi coloré suivant la propagation de l'information ce qui permet, en théorie, de traiter les flux d'informations indirects.

Très populaire dans le domaine de l'analyse de malware [34], cette technique est parfois utilisée pour garantir l'application d'objectifs de sécurité. Par exemple les auteurs de [35–37] s'appuient sur l'approche Blare [38, 39] pour garantir la cohérence d'accès aux objets de la Machine virtuelle Java ou Dalvik vis à vis d'une politique de sécurité centrée sur les flux directs. Pour cela, ils demandent de spécifier une politique pour chaque objet de la JVM et, si besoin, une coloration initiale de sorte que tout accès indirect qui ne serait pas autorisé par la politique d'accès direct soit considéré comme malveillant et donc illégitime.

Blare est représentatif de ce type d'approche où la logique de contrôle est basée sur le fait qu'un sujet d'une couleur donnée ne peut accéder qu'aux objets de la même couleur que lui. On remarque immédiatement que le nombre de couleurs utilisées influence directement la granularité de contrôle. Ainsi si le nombre de couleurs utilisées est trop faible, il devient difficile de distinguer avec précisions certains flux d'informations. À l'opposé, si le niveau de coloration est trop élevé, la propagation des couleurs sera telle que chaque entité sera progressivement coloriée avec les mêmes couleurs, annihilant ainsi la possibilité de garantir l'application de propriétés de sécurité. Une mauvaise coloration conduit alors à une sur-approximation/sous-approximation des flux d'informations légitimes et induit donc des faux-positifs/faux-négatifs.

Prenons l'exemple de JBlare, l'implémentation de Blare pour Java [35]. Ses auteurs illustrent le cas d'un serveur web compromis qui tente d'accéder au répertoire personnel des utilisateurs du système. Ces répertoires et le serveur web étant colorés différemment, Blare détecte la violation de la politique et bloque donc l'opération d'accès. Cependant la politique Blare n'utilise que deux couleurs dans cet exemple, *bleu* et *jaune* respectivement. Ainsi, il n'est pas possible d'autoriser le serveur web à accéder de façon légitime au dossier d'un utilisateur qui souhaite le partager. Dans ce cas, JBlare lève un faux-positif lorsque le serveur web accède au dossier partagé. Par ailleurs, JBlare autorise implicitement

toutes opérations entre les répertoires personnels des utilisateurs du système ce qui n'est pas nécessairement légitime (cas des répertoires `/root` et `/home/user`). Pour corriger ce défaut, il faudrait donc augmenter considérablement le nombre de couleurs avec au moins une couleur par utilisateur, et là encore le partage entre utilisateurs est problématique. Dans [40], ils visent à permettre un partage via des processus multicouleurs, cependant cette approche tend soit à tout colorer par des couleurs multiples soit à ce que les couleurs multiples soient inaccessibles pour éviter la contamination.

Les travaux initiés par [41] proposent un moniteur de référence pour la machine virtuelle Java basé sur la coloration [42], idée reprise par [32], avec les limitations en termes de performance et d'expressivité des politiques que cela implique.

La littérature en vigueur comme par exemple [43] s'accorde à dire que la complexité des algorithmes de coloration et de vérification est telle qu'il est obligatoire de réduire les couleurs disponibles pour ne pas nuire aux performances. Ainsi, beaucoup d'implémentations basées sur la coloration doivent se limiter afin de passer à l'échelle. De plus, elles sur-approximent les flux d'informations car elles n'offrent pas de possibilité de logique de contrôle avancée comme le permet JAAS. En conséquence, toutes les implémentations des techniques de coloration pour Java ou Android, telles que JBlare [35] et AndroBlare [36, 37], souffrent de ce défaut.

Au final, les approches par coloration ne sont applicables que pour des petits programmes tels qu'un malware et ne conviennent pas à des applications de plus grande envergure. De plus et bien qu'il soit possible de définir la politique de sécurité, c'est à dire les relations entre couleurs, le langage utilisé permet difficilement d'exprimer un large panel d'objectifs de sécurité.

### 2.4.4 Prédicats de sécurité

Un prédicat de sécurité est une fonction booléenne qui évalue les autorisations d'accès aux endroits clefs d'un programme. Son principal avantage est qu'il apporte une preuve formelle de l'application des propriétés de sécurité qu'il garanti.

Les cartes à puce sont le domaine d'application de prédilection de ce type d'approche [44]. On note toutefois certains travaux de recherche pour des applications plus génériques tels que Haskell [45], Caml [46] ou Java [47].

En dehors des problèmes liés aux contraintes de licence et au contournement des mesures de protection [48] pour les applications non ouvertes, ce type d'approche

requiert en pratique un long travail d'instrumentation qui nécessite d'avoir accès au code source de l'application. De plus, elles sont proches des méthodes de vérification [49,50] et ne sont applicables que dans des environnements restreints.

Ainsi, ce type d'approche est lié à du développement de code sûr. Cela pose des problèmes de passage à l'échelle et n'est pas accessible à un administrateur système qui souhaiterait déployer des applications Java ou du code mobile.

### 2.4.5 Contrôle d'accès

Le contrôle d'accès s'intéresse principalement à garantir deux propriétés de sécurité : la confidentialité et l'intégrité des informations contenues dans le système. Pour cela, l'approche consiste à vérifier qu'une entité *Sujet* (généralement un utilisateur) possède les droits nécessaires lorsque celle-ci souhaite accéder à une entité *Objet* (généralement une ressource du système). La définition de la politique ainsi que le contrôle peuvent se faire soit de façon discrétionnaire (les droits d'accès sont à la "discrétion" de l'utilisateur) soit de façon obligatoire (indépendamment de la volonté de l'utilisateur). Les approches discrétionnaires sont réputées pour être plus faibles que les approches obligatoires [51]. C'est pourquoi nous nous intéressons ici qu'aux seules approches obligatoires disponibles pour Java.

#### Contrôle d'accès multi-niveaux/multi-catégories

Historiquement, les approches multi-niveaux (MLS) sont l'application à l'informatique des principes de classification de l'information issus du domaine militaire. L'idée fondamentale est d'attribuer à chaque entité et ressource du système un niveau de sensibilité (SL) puis de réaliser une comparaison arithmétique pour en déduire les autorisations. La politique de sécurité correspondante est alors codée en dur dans l'implémentation MLS et découle directement de la propriété de sécurité à garantir. Par exemple, les implémentations de type Bell-Lapadula [52, 53], Biba [54] et Mandatory Integrity Control (MIC) [55] ont respectivement pour but de garantir la confidentialité, l'intégrité et la disponibilité (cf. table 2.1).

Les approches multi-catégories (MCS) sont orthogonales à MLS. Elles sont basées sur l'utilisation de catégories de sensibilité (SC) attribuées à chaque entité et ressource du système. Les autorisations sont ensuite calculées en fonction des relations qui existent entre les catégories du Sujet et celles de l'Objet.

Bien que souvent limités à la garantie d'une seule propriété de sécurité, les modèles de type MLS/MCS ont cependant l'avantage d'être performants et

| | Bell-Lapadula (confidentialité) | Biba (intégrité) | MIC (disponibilité) |
|---|---|---|---|
| no-read-up | ✓ | | ✓ |
| no-write-up | | ✓ | ✓ |
| no-execute-up | ✓ | | ✓ |
| no-read-down | | ✓ | |
| no-write-down | ✓ | | |
| no-execute-down | | ✓ | |

TABLE 2.1 – Principales politiques de sécurité MLS

faciles à implémenter car ils ne font intervenir que des opérations booléennes. Les auteurs de [56] proposent ainsi de propager le label MLS/MCS des contextes SELinux dans les applications Java ; le but étant de renforcer l'isolation des flux d'informations "en transit" dans la JVM. Mais cela ne suffit pas à garantir un contrôle fin de tous les flux d'information d'une application car la politique MLS appliquée reste limitée aux seules règles de la littérature.

Autre exemple notable, Laminar [57] s'appuie sur une logique MLS pour contrôler les flux d'informations à la fois au niveau système et au niveau applicatif. Il s'agit ici d'un modèle de contrôle des flux décentralisé (DIFC) [58] duquel s'inspire des implémentations de contrôle au niveau langage [59] et système [60]. Le principe est de considérer que la signification d'un niveau de sécurité est identique au niveau système et applicatif. Cette hypothèse permet une approche globale pour contrôler tous les flux d'informations. Néanmoins, Laminar impose de réécrire les applications à protéger et ne tient pas compte des propriétés de sécurité exprimées par les politiques Java ce qui limite grandement l'utilité pratique de cette approche. Plus généralement, les approches de type DIFC sont connues pour induire une sur-approximation [61] des flux d'information ce qui limite encore plus leur intérêt pratique.

En pratique, les modèles MLS autorisent une entité à changer temporairement de niveau pour répondre à des contraintes fonctionnelles comme par exemple les mécanismes de déclassification de l'information dans les implémentations de type Bell-Lapadula ou bien le User Access Control (UAC) de Windows NT6 [18]. A terme, cela revient à faire migrer progressivement toutes les entités du système vers le niveau de sécurité minimal (Bell-Lapadula) ou maximal (Biba, MIC), rompant ainsi l'isolation offerte par la protection.

#### Contrôle d'accès basé sur les types

Le contrôle d'accès basé sur les types (TE) propose d'étiqueter chaque entité et ressource du système avec un identifiant défini par politique. La politique établit les règles d'accès entre types ce qui permet de factoriser les règles de contrôle

18. Windows Vista, Windows Seven, Windows 8

pour les entités de même nature (i.e. ayant le même type). Cette étiquette (ou label) est extensible et peut contenir des informations complémentaires comme, par exemple, des attributs RBAC et MLS/MCS dans le cas d'un label SELinux [62]. Le Domain & Type Enforcement (DTE) étend l'approche TE en tenant compte des cas d'usage des entités et ressources du système [63].

Le principal avantage de ces approches est que la granularité du contrôle d'accès peut être ajustée en faisant varier la granularité des règles de typage. De même, en contrôlant les relations entre types il est possible d'ajuster les autorisations. Il s'agit d'une approche fine avec comme défaut principal la difficulté de calcul des politiques.

Les auteurs de [64, 65] proposent d'étendre le contrôle d'accès SELinux du système à Java. Cependant, la machine virtuelle Java n'implémente pas de moniteur de référence capable d'appliquer ce type de politique. En conséquence, ces approches se limitent à contrôler les flux entrant et sortant de la JVM.

L'inconvénient majeur de l'approche DTE est que la richesse d'expressivité du langage induit obligatoirement une complexité des politiques de sécurité qui est bien au delà des capacités humaines. En pratique, une politique DTE est donc soit très permissive soit trop restrictive ce qui nécessite des politiques complémentaires [4, 66]. Même avec deux niveaux de politique, le problème de la complexité de leurs définitions reste entier et ces approches statiques couvrent mal des besoins dynamiques contrairement à des systèmes extensibles comme JAAS.

Les auteurs de [67] proposent de contrôler les sessions HTTP par une approche de type TE afin de renforcer l'isolation des flux d'informations dans la couche métier des services web. L'originalité de l'approche est que les règles de contrôle d'accès peuvent être calculées automatiquement en fonction de la spécification des processus métiers [3]. Il s'agit donc à la fois d'une approche fine du type DTE mais dont les politiques sont calculables automatiquement pour des processus métiers dont le code est généré dynamiquement. En conséquence, le calcul des politiques TE se trouve complètement automatisé et respecte les objectifs de sécurité des processus métiers.

## 2.5 Conclusion et présentation du besoin

Nous avons présenté les modèles d'attaques existants sur Java en distinguant la corruption de l'espace mémoire Java incluant la confusion de type qui permet de violer l'intégrité de l'espace Java et les défauts de contrôle d'accès qui conduisent à un contournement de JAAS et la désactivation de l'inspection de pile. Nous

pensons que ces deux types d'attaques peuvent être prévenus en réalisant un contrôle fin des privilèges entre objets.

JAAS propose un modèle solide et extensible avec la possibilité de définir de nouvelles capacités, une logique avancée et ouverte de contrôle des capacités. Le problème essentiel de JAAS est la fragilité de son moniteur de référence car reportant certaines responsabilités sur le programmeur. Une des conséquences est que le contournement du contrôle, réalisé par JAAS, des élévations de privilèges est rendu aisé.

Des approches complémentaires visent des alternatives à JAAS. Elles sont basées sur l'analyse de programme ou des prédicats, sur la coloration ou sur un contrôle d'accès obligatoire. Les deux premières classes de solution n'offrent pas la possibilité de moduler les objectifs de sécurité et présentent des problèmes de passage à l'échelle. Les approches obligatoires sont peu développées dans le monde Java. Il manque notamment des solutions du type DTE pour contrôler finement les interactions entre les objets Java. Cependant, le problème de complexité de calcul des politiques DTE est de façon générale ouvert mais des travaux montrent que dans certains cas ce problème peut être résolu.

Au final, notre étude vise donc à proposer un modèle du type DTE qui manque pour Java en montrant son efficacité sur des cas pratiques de sécurité. Ainsi, nous montrerons qu'une telle approche peut être pertinente pour améliorer la sécurité des applications Java. Nous essaierons de proposer une solution pour résoudre le problème du calcul des politiques DTE afin d'obtenir une approche autonome qui supporte *a minima* le modèle JAAS. Ainsi, nous couvrirons trois besoins essentiels, un modèle de contrôle fin du type DTE, une façon d'automatiser et optimiser les politiques et une compatibilité complète avec le modèle JAAS puisque celui-ci est largement utilisé en pratique de part son extensibilité.

| | Efficacité | Passage à l'échelle | Maintenabilité |
|---|---|---|---|
| JAAS | ++ | +++ | ++ |
| Analyse de programme | - - - ① | - ③ | ++ |
| Coloration dynamique | + | - - ② | + |
| Approche formelle | ++ | - - - ② | - - ③ |
| Contrôle d'accès DTE | ++ | +++ | - - ④ |
| Notre objectif | +++ | +++ | +++ |

TABLE 2.2 – Notre point de vue général sur l'état de l'art.

① L'analyse de programme est prouvée comme faible (cf. section 2.4.2) ; ② La coloration dynamique et les approches formelles ont une complexité algorithmique limitant leur applicabilité (cf. sections 2.4.3 et 2.4.4) ; ③ Cette dernière avec l'analyse de programme nécessite un accès direct au code source du programme ce qui n'est pas toujours possible (cf. sections 2.4.2 et 2.4.4) ; ④ Les approches de type DTE sont limitées par la difficulté de définir et maintenir des politiques de sécurité (cf. section 2.4.5).

Chapitre 3

# Formalisation d'un modèle à objets pour exprimer des règles et politiques de sécurité

## Sommaire

> *La clef pour concevoir de grands systèmes extensibles réside plus dans la façon dont ses modules communiquent que sur les données et comportements internes de ces modules.*
>
> Alan Kay [68]

Notre état de l'art montre qu'il n'existe pas de modèle de protection obligatoire satisfaisant pour Java. Le fait que JAAS s'appuie sur des capacités pour contrôler les flux d'informations entre objets Java nous semble pertinent. Mais, en pratique, l'implémentation par JAAS de l'inspection de la pile d'exécution ne permet pas de contrôler efficacement les élévations de privilèges. Plus globalement, la question est de savoir comment contrôler les privilèges entre objets pour que la protection soit plus efficace que JAAS et de type obligatoire. L'approche générale proposée doit bien sûr s'appliquer à Java, mais nous souhaitons aussi un cadre conceptuel plus large afin que l'approche soit extensible à des systèmes à objets autre que Java.

C'est pourquoi, ce chapitre propose un modèle de contrôle général pour les systèmes à objets tels qu'ils ont été initialement définis par les travaux de Ole-Johan Dahl, Kristen Nygaard et Alan Kay [69]. Le concept de l'objet est apparu au milieu des années 60 avec le langage Simula et s'applique aujourd'hui à des domaines variés tels que les systèmes distribués, la modélisation ou encore le génie logiciel (cf. tableau 3.1). De tous les environnements à objets qui existent, nous nous intéressons principalement à celui des langages de programmation (POO) et plus particulièrement aux langages basés sur les classes où figure Java. Nous traiterons également le cas des environnements à prototypes même si ce paradigme est très éloigné de notre problématique principale qui est Java et les systèmes à objets basés sur les classes. De même, nous montrerons que le modèle de contrôle que nous proposons est généralisable à des systèmes à objets répartis basés sur le principe des appels de procédures distantes (RPC).

| Objets répartis | Langages à classes | Langages à prototypes |
|---|---|---|
| Java RMI | Java | Javascript |
| CORBA | C++ | Self |
| SOAP | UML | Python |
| REST | PHP | |
| RPC | | |

TABLE 3.1 – Exemples de systèmes à objets

Par nature, les environnements orientés objets (OO) ne contiennent que des objets. Les notions communes de système d'exploitation telles que *thread*, *processus*, *fichier* ou *appel système* sont représentées par des objets. Ainsi en contrôlant les relations entre objets il semble possible de garantir des propriétés de sécurité

à la fois pour les objets métiers et les objets du système d'exploitation. Contrairement aux approches telles que SELinux qui se limitent aux seuls objets du système d'exploitation nous proposons une approche plus profonde, c'est à dire permettant aussi le contrôle entre objets internes à un processus.

Nous faisons donc l'hypothèse que le contrôle doit porter sur les objets et leurs relations. Cela implique de disposer d'une modélisation des systèmes à objets et de leurs relations. Il existe de nombreux travaux dans la littérature pour modéliser les systèmes à objets dont la notation UML [70–72] ou les approches formelles tels que [73, 74]. Néanmoins, les modèles proposés ont souvent pour but la conception ou la vérification de systèmes. Ils sont donc, de fait, soit trop graphiques, soit trop abstraits, soit trop détaillés comme [75] qui s'appuie sur 400 pages de modélisation. Il est donc nécessaire de proposer une modélisation plus adaptée au contrôle basé sur un modèle général d'objets et sur les relations entre ces objets.

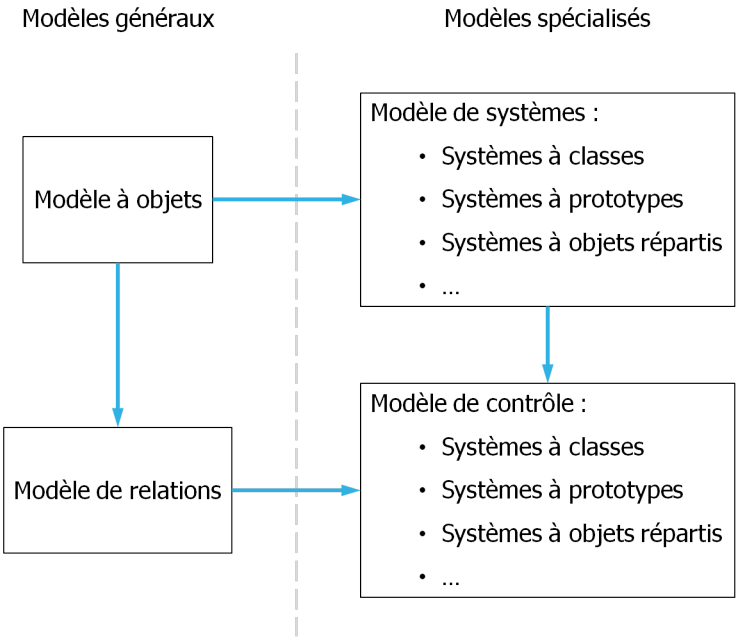



FIGURE 3.1 – Approche globale et organisation de ce chapitre

Ainsi, nous commençons ce chapitre par une modélisation générale des éléments d'un système à objets en nous appuyant principalement sur une notation ensembliste. Puis, nous étendrons ce modèle avec les éléments propres à trois types de système à objets : les langages à classes, les langages à prototypes et les systèmes à objets répartis (cf. tableau 3.1). Ensuite, nous formaliserons les différentes relations entre les objets afin d'en déduire un modèle de contrôle pour chaque type de système considéré. La figure 3.1 représente graphiquement le raisonnement global suivi par ce chapitre.

## 3.1 Modèle général de système à objets

Un *objet* doit être vu comme une entité indépendante établissant des relations avec les autres objets. Par exemple considérons deux objets : *porte* et *individu*. Ces deux objets sont en relation car un *individu* peut ouvrir la *porte*. Pour cela, il pourra accéder à des attributs de la porte telle que la poignée et à des méthodes telles que *appuyer_sur_poignée()* et *tirer_porte()*. Nous proposons donc ici une notation pour les différents éléments constituant les objets. Dans la section suivante nous définirons les relations entres objets au moyen de la notation définie dans cette section.

De façon générale, il existe des données de nature élémentaire que l'on qualifie de type primitif. À partir de types primitifs on construit des types plus élaborés appelés types structurés car constitués de types primitifs ou structurés. L'objet est par définition une entité de type structuré dont chaque élément, appelé membre, contient différents éléments primitifs. Nous distinguons dans cette section deux types de membres que sont le champ et la méthode.

Par exemple, en langage Java, l'ensemble des types primitifs comprend l'entier, le booléen, le flottant, le caractère, l'énumération et la référence. De même l'objet `java.lang.Integer` est principalement constitué d'un champ du type primitif "entier" et de méthodes. La suite de cette section formalise ces différentes notions : les types primitifs ; les objets ainsi que leurs noms et leurs types ; les membres des objets comprennent les champs et les méthodes. Nous proposons aussi la notion de localisation des objets, celle-ci permettant ensuite de définir la notion de référence qui est à la base des relations entre objets. Dans la formalisation que nous proposons, comme illustrée par le dessin (cf. figure 3.2), on voit que tous les membres d'un objet sont aussi des objets.

### 3.1.1 Objets et représentation d'objets

**Notation 3.1 (Objets)** *Soit $O$ l'ensemble des objets du système.*

*Ainsi tout objet $o_k$ du système appartient à l'ensemble $O$, $o_k \in O$.*

Dans la notation UML, on représente graphiquement un objet à l'aide d'un rectangle où le nom et le type de l'objet apparaissent soulignés. Néanmoins UML offre des détails de modélisation qui ne nous intéressent pas et, à l'inverse,

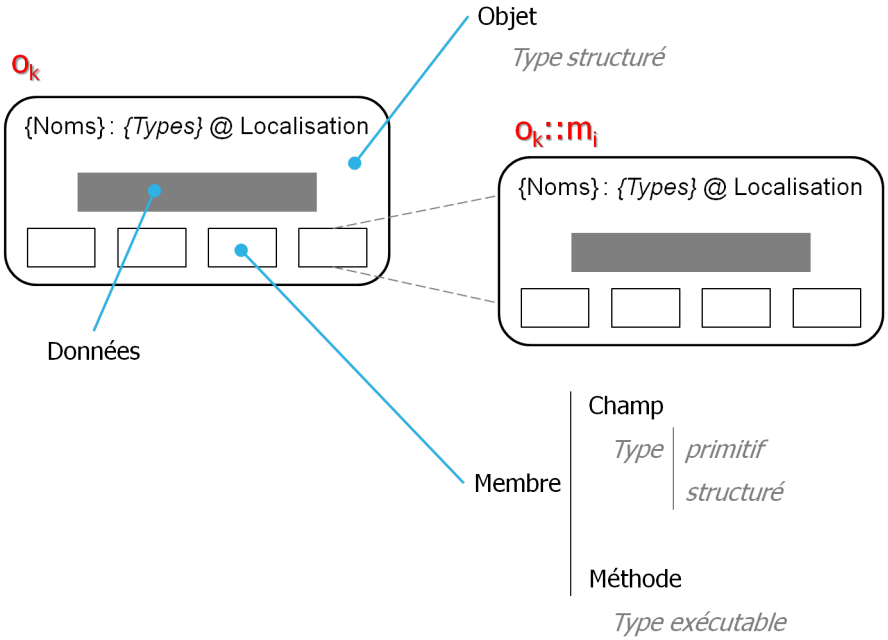



FIGURE 3.2 – Représentation graphique des éléments structurant d'un objet

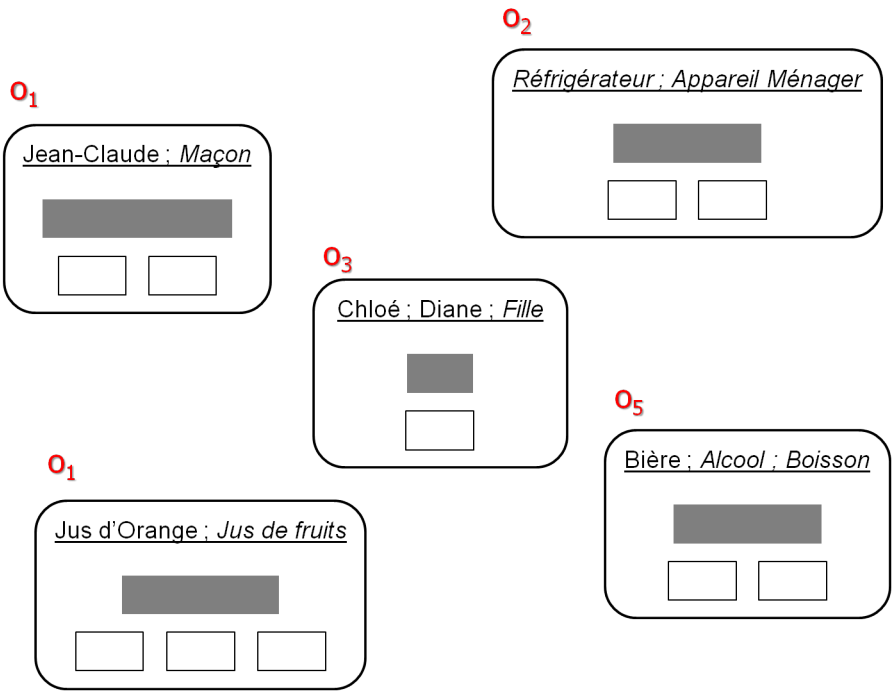



FIGURE 3.3 – Ensemble de cinq objets

Ce système est composé de 5 objets. c'est à dire $O = \{o_1, o_2, o_3, o_4, o_5\}$

sont trop flous notamment sur les noms et les types qu'un objet peut posséder. C'est pourquoi nous proposons une représentation graphique plus adaptée car plus précise notamment sur les noms, types et localisation d'objet (cf. figure 3.3). Nous détaillons ces notions dans la suite du document.

### 3.1.2 Noms des objets

Dans la vie courante, nos noms et prénoms permettent de nous désigner indépendamment de notre localisation géographique. Il en va de même pour les objets qui, comme nous, peuvent posséder un ou plusieurs noms. Bien entendu, le nom d'un objet n'est pas unique mais l'ensemble des noms d'un objet participe à la désignation de celui-ci. Il est possible de définir des relations entre objets, par exemple entre *Fille* et *Réfrigérateur*. Ces relations doivent pouvoir s'exprimer au moyen des noms mais aussi servir de politique de contrôle d'accès. C'est pourquoi nous définissons l'une des caractéristiques d'un objet qui est l'ensemble de ses noms.

**Notation 3.2 (Noms)** *Soit $O$ un environnement à objets et $N$ l'ensemble des noms d'objets du système. On définit l'application Noms qui pour un objet $o_k \in O$ renvoie le(s) nom(s) de l'objet $o_k$ tel que :*

$\forall o_k \in O, \exists\, noms_k \subset N,$

$$\begin{aligned} Noms\colon O &\longrightarrow N \\ o_k &\longmapsto \quad noms_k \equiv \left\{nom_1^k, nom_2^k, \cdots, nom_n^k\right\} \end{aligned}$$

Dans notre modèle et contrairement à la notation UML où les objets ne peuvent avoir au maximum que deux noms, nous considérons un ensemble plus large. Selon notre notation, le(s) nom(s) d'un objet est une chaîne de caractères qui apparaît soulignée dans la représentation graphique de l'objet. Par exemple, si nous reprenons les objets de la figure 3.3, nous pouvons calculer :

$$\begin{aligned} O \equiv& \{o_1, o_2, o_3, o_4, o_5\} \\ Noms(o_1) \equiv& \{\text{"Jean-Claude"}\} \\ Noms(o_2) \equiv& \emptyset \\ Noms(o_3) \equiv& \{\text{"Chloé", "Diane"}\} \\ Noms(o_4) \equiv& \{\text{"Jus d'Orange"}\} \\ Noms(o_5) \equiv& \{\text{"Bière"}\} \end{aligned}$$

Nous pourrions ainsi exprimer une politique autorisant ou non une relation entre *Chloé* et *Jus d'orange*.

### 3.1.3 Types des objets

L'objet est, par définition, une entité qui regroupe des données et les traitements sur ces données ; chaque objet pouvant éventuellement être en rapport avec une chose tangible du monde réel. Calculer le type d'un objet permet selon nous de connaître la nature de cet objet et d'en déduire ses privilèges. Par ce biais, il est possible d'exprimer une politique de sécurité qui tient compte des types des objets.

Le type d'un objet peut être déduit des données et méthodes qui le compose [1]. Par exemple, si un objet Python définit les méthodes *___getitem___(self, key)* et *___setitem___(self, key, item)* alors cet objet contient des données indexées, c'est à dire qu'il s'agit d'un tableau ou d'une table de hachage, par exemple. À l'inverse, le type d'un objet caractérise la structure de l'objet et la manière de l'interpréter. Par exemple, un objet de type *CoordonnéeCartésienne* doit définir au moins deux membres : *x_axis* et *y_axis*.

**Notation 3.3 (Types)** *Soit $O$ un environnement objets et $T$ l'espace des types du système. On définit l'application Types qui pour un objet $o_k \in O$ renvoie le(s) type(s) de l'objet $o_k$ tel que :*

$\forall o_k \in O,\ \exists\, types_k \subset T,$

$$\begin{aligned} Types\colon O &\longrightarrow T \\ o_k &\longmapsto \quad types_k \equiv \left\{type_1^k, type_2^k, \cdots, type_t^k\right\} \end{aligned}$$

Selon la figure 3.3 ci-avant, nous savons que l'objet $o_1$ est associé au nom générique "Maçon" qui traduit le type auquel $o_1$ appartient. Cela signifie alors que $Types(o_1) \equiv \{\text{"Maçon"}\}$, ou plus simplement $Types(\text{"Jean-Claude"}) \equiv \{\text{"Maçon"}\}$. Bien entendu, tous les "Jean-Claude" ne sont pas forcément "Maçons" et tous les

1. "Si ça ressemble à un canard, si ça nage comme un canard et si ça cancane comme un canard, c'est qu'il s'agit sans doute d'un canard." - James Whitcomb Riley (1849-1916), postulat du test du canard repris par Alan Turing dans [76]

"Maçons" ne s'appellent pas "Jean-Claude". L'objet $o_1$ représente juste un "Maçon" particulier qui se nomme "Jean-Claude". Plus généralement, nous pouvons calculer :

$$\begin{aligned} O \equiv & \{o_1, o_2, o_3, o_4, o_5\} \\ Types(o_1) \equiv & \{\text{"Maçon"}\} \\ Types(o_2) \equiv & \{\text{"Réfrigérateur", "Appareil Ménager"}\} \\ Types(o_3) \equiv & \{\text{"Fille"}\} \\ Types(o_4) \equiv & \{\text{"Jus de Fruit"}\} \\ Types(o_5) \equiv & \{\text{"Alcool", "Boisson"}\} \end{aligned}$$

Il est à remarquer que notre modélisation considère qu'un objet peut avoir plusieurs types, par exemple via la coercition. Mais cette considération apporte surtout des facilités de modélisation que nous verrons par la suite.

```
1 java.lang.Object obj = "toto tata tutu";    /* cast String vers Object */
2 java.lang.String str = (java.lang.String)obj; /* cast Object vers String */
```

FIGURE 3.4 – Exemple de coercition en Java.

La chaîne de caractères à droite de la première affectation est un objet Java de type *String*. Celui-ci est successivement affecté à deux variables distinctes via une coercition de *String* vers *Object* puis de *Object* vers *String*. Cela revient pour *obj* à avoir deux types : `java.lang.Object` et `java.lang.String`.

### 3.1.4 Types primitifs

Le contenu d'une donnée, c'est à dire sa valeur, doit être interprété selon son type qui peut être soit primitif soit structuré. Les types primitifs servent à créer des données de base à l'aide de littéraux. Par exemple, l'affectation `int value = 4` correspond à la création d'une donnée entière dont la valeur est 4.

**Notation 3.4 (Types primitifs)** *On définit $T_{primitif} \subset T$ l'ensemble des types primitifs du système et $prim_k$ un élément de cet ensemble.*

Par exemple, les types *number*, *boolean* et *string* sont les seuls types primitifs définis par les spécifications Javascript. C'est à dire que l'ensemble des types Javascript primitifs est $T_{primitif} \equiv \{"number", "boolean", "string"\}$. À l'inverse, les types *Number* et *String* ne sont pas primitifs en Java.


FIGURE 3.5 – Comment différentier ces deux objets ?

Ce système est composé de deux objets réels identiques mais parfaitement distincts ; c'est à dire $O \equiv \{o_1, o_2\}$. Pour différentier l'objet $o_1$ de l'objet $o_2$, il suffit de regarder où chacun d'entre eux se situent. C'est à dire, calculer *Localisation($o_1$)* et *Localisation($o_2$)*. Ainsi, si $Localisation(o_1) \equiv$ "*droite*" et $Localisation(o_2) \equiv$ "*gauche*" alors nous savons que le feutre de gauche est l'objet $o_2$ et celui de droite est l'objet $o_1$.

### 3.1.5 Localisation d'un objet

Chaque objet, qu'il soit concret, abstrait ou virtuel, possède une existence physique et réside à un endroit précis de son environnement. L'emplacement physique d'un objet est par essence unique car deux objets distincts ne peuvent exister à la fois au même endroit. Pour s'en convaincre, nous pouvons prendre l'exemple de deux objets a priori distincts comme ceux illustrés sur la figure 3.5. Si les informations de localisation de ces deux objets sont identiques alors nous avons la garantie qu'il s'agit, en réalité, d'un seul et même objet. Dans le cas contraire, il s'agit effectivement de deux objets distincts. Ainsi, la localisation d'un objet nous apporte la garantie d'unicité de l'objet contrairement à son nom. Il est important d'en tenir compte puisque cette information unique permet un contrôle précis.

**Notation 3.5 (Localisation)** *Soit $O$ un système à objets et $L$ l'espace des emplacements d'objet possibles dans $O$.*

*On définit l'application Localisation qui pour un objet $o_k \in O$ renvoie l'emplacement dans $O$ de l'objet $o_k$ telle que :*

$$\forall o_k \in O,\ \exists!\ loc_k \in L,\ Localisation\colon O \longrightarrow L$$
$$o_k \longmapsto loc_k$$

En pratique, l'application *Localisation* n'est pas bijective car un emplacement donné ne contient pas nécessairement un objet. Par contre, un objet possédera toujours une localisation.

### 3.1.6 Membres d'objet et valeurs

L'objet définit des éléments qui renseignent sur le(s) type(s) et la localisation de chaque donnée qui le compose. Ces éléments structurant de l'objet sont les *membres* de l'objet. Dans la pratique, un membre est un objet dont la localisation se trouve à l'intérieur de l'objet auquel le membre appartient. La notion de membre est importante selon nous car celle-ci rentre en jeux dans les relations entre objets et le fait de pouvoir qualifier un membre précis d'un objet autorise la définition de règles de contrôle à grain fin.

**Notation 3.6 (Membres)** *Soit $O$ un système à objets.*

1. *On définit l'application Membres qui pour un objet $o_k \in O$ renvoie les objets membres $m_i^k \in M$ de l'objet $o_k \in O$ telle que :*
   $\forall o_k \in O, \exists\, membres_k \subset M,$

$$\begin{aligned} Membres\colon O &\longrightarrow M \\ o_k &\longmapsto \quad membres_k \equiv \{m_1^k, m_2^k, \cdots, m_j^k\} \end{aligned}$$

2. *On note $o_k :: m_i$ un membre particulier de l'objet $o_k$ tel que :*
   $\forall o_k \in O, \exists! m_i^k \in Membres(o_k),\ o_k :: m_i \equiv m_i^k$

```
1  typedef struct {
2      uint32_t ip_version             : 4;
3      uint32_t internet_header_length : 4;
4      uint32_t type_of_service        : 8;
5      uint32_t total_length           : 16;
6
7      uint32_t id                     : 16;
8      uint32_t flags                  : 3;
9      uint32_t fragment_offset        : 13;
10
11     uint32_t time_to_live           : 8;
12     uint32_t protocol               : 8;
13     uint32_t checksum               : 16;
14
15     uint32_t source_address         : 32;
16     uint32_t destination_address    : 32;
17
18     uint32_t options[];
19   } ip_header_t;
20
21 ip_header_t ip_packet;    /* --> o_1 */
```

FIGURE 3.6 – Exemple d'implémentation en C d'une entête de paquet IP.

Par exemple, la structure C de la figure 3.6 contient treize membres, tous de type primitif. La dernière ligne de ce programme montre la déclaration d'un

objet de type *ip_header_t* qui est donc constitué de treize membres ; soit un total de quatorze objets en tout pour le système considéré. En pratique, la localisation de chacun des membres de cet objet *ip_header_t* est une distance qui sépare le membre considéré du premier octet de l'objet (c-à-d un *offset*). Ainsi, le membre *id* de cette structure indique que le membre qui contient la donnée d'identification d'un paquet IP se situe $4+4+8+16 = 32$ bits après le premier bit de l'objet. Plus particulièrement, nous pouvons alors calculer :

$$O \equiv \{o_1, o_2, o_3, o_4, o_5, o_6, o_7, o_8, o_9, o_{10}, o_{11}, o_{12}, o_{13}, o_{14}\}$$

$$\begin{aligned}
Noms(o_1) &\equiv \{\text{"}ip_packet\text{"}\} \\
Types(o_1) &\equiv \{\text{"}ip_header_t\text{"}\} \\
Membres(o_1) &\equiv \{m_1^1 = o_2, m_2^1 = o_3, \cdots, m_{13}^1 = o_{14}\}
\end{aligned}$$

$$\begin{aligned}
Noms(m_1^1) &\equiv Noms(o_1 :: m_1) \equiv Noms(o_2) \equiv \{\text{"}ip_version\text{"}\} \\
Types(m_1^1) &\equiv Types(o_1 :: m_1) \equiv Types(o_2) \equiv \{\text{"}uint32_t\text{"}\} \\
Localisation(m_1^1) &\equiv Localisation(o_1) + 0\ bits \\
Membres(m_1^1) &\equiv \emptyset
\end{aligned}$$

$$\begin{aligned}
Noms(m_2^1) &\equiv Noms(o_1 :: m_2) \equiv Noms(o_3) \equiv \{\text{"}internet_header_length\text{"}\} \\
Types(m_2^1) &\equiv Types(o_1 :: m_2) \equiv Types(o_3) \equiv \{\text{"}uint32_t\text{"}\} \\
Localisation(m_2^1) &\equiv Localisation(o_1) + 4\ bits \\
Membres(m_2^1) &\equiv \emptyset \\
&\vdots
\end{aligned}$$

Ainsi, c'est la notion de membre qui définit comment sont assemblées les données d'un objet. Un objet qui ne contient qu'une seule et unique donnée primitive doit être considéré comme un cas particulier. On qualifie ce genre d'objet d'enveloppe pour un type primitif ou, plus simplement, *objet enveloppe*.

**Notation 3.7 (Valeurs)** *Soit $O$ un système à objets.*

1. *On définit les ensembles $V[prim_k]$ constitués des valeurs possibles pour une donnée primitive de type $prim_k \in T_{primitif}$.*
   *Exemples pour Javascript :*

$$\begin{aligned}
V[number] &\equiv \{1, 2, \sqrt{7}, \pi, \cdots\} \equiv \mathbb{R} \\
V[boolean] &\equiv \{true,\ false\} \\
V[string] &\equiv \{\text{"}a\text{"}, \text{"}abcd\text{"}, \text{"}baba\text{"}, \cdots\}
\end{aligned}$$

2. *On définit l'application Valeur qui pour un objet $o_k \in O$ renvoie l'ensemble des valeurs primitives $v_i^k \in V$ composant les données de l'objet $o_k$ telle que : $\forall o_k \in O$, $\exists\ valeur_k \subset V$,*

$$\begin{aligned} Valeur\colon O &\longrightarrow V \\ o_k &\longmapsto \ valeur_k \equiv \{v_i^k | v_i^k \in V[T_{primitif}]\} \end{aligned}$$

Prenons l'exemple du tableau d'entiers `int array[]={ 7, 2, 4, 9 }`. Un tableau n'étant pas un type primitif, il s'agit donc d'un objet contenant des données indexées où chaque membre est un objet enveloppe de type *entier*. Notre notation nous permet alors de connaître la valeur de chacun de ces membres, c'est à dire les données que contient le tableau :

$$O \equiv \{o_1, \cdots\}$$

$$\begin{aligned} Noms(o_1) &\equiv "array" \\ Types(o_1) &\equiv \{"tableau", "entier"\} \end{aligned}$$

$$\begin{aligned} Membres(o_1) &\equiv \{m_1^1, m_2^1, m_3^1, m_4^1\} \\ Types(m_1^1) &\equiv "entier", \quad Valeur(m_1^1) \equiv v_1^1 = 7 \\ Types(m_2^1) &\equiv "entier", \quad Valeur(m_2^1) \equiv v_2^1 = 2 \\ Types(m_3^1) &\equiv "entier", \quad Valeur(m_3^1) \equiv v_3^1 = 4 \\ Types(m_4^1) &\equiv "entier", \quad Valeur(m_4^1) \equiv v_4^1 = 9 \end{aligned}$$

$$\begin{aligned} Valeur(o_1) &\equiv \{v_1^1, v_2^1, v_3^1, v_4^1\} \\ &\equiv \{7, 2, 4, 9\} \end{aligned}$$

### 3.1.7 Champs et méthodes d'objet

Les membres d'un objet peuvent être rangés en deux ensembles. Le premier ensemble correspond aux *champs*[2], c'est à dire les membres qui contiennent de l'information. Le second ensemble est celui des *méthodes*, c'est à dire des membres qui contiennent du code.

Certains systèmes à objets font une distinction franche et claire entre les *champs* et les *méthodes* comme PHP ou Java par exemple. A contrario, d'autres systèmes

2. ou "attribut" pour les francophones.

tels que Javascript ou Python considèrent que les notions de *champs* et de *méthodes* n'ont de sens qu'au moment de leur utilisation.

**Notation 3.8 (Champs)** *Soit $O$ un système à objets.*

*On définit l'application Champs qui pour un objet $o_k \in O$ renvoie les champs $a_i^k \in A \subset M$ de l'objet $o_k \in O$ telle que :*
$\forall o_k \in O$, $\exists\, champs_k \subset membres_k$,

$$\begin{aligned} Champs\colon O &\longrightarrow A \\ o_k &\longmapsto champs_k \equiv \{a_1^k, a_2^k, \cdots, a_j^k\} \end{aligned}$$

**Notation 3.9 (Méthodes)** *Soit $O$ un système objets.*

*On définit l'application Méthodes qui pour un objet $o_k \in O$ renvoie les méthodes $f_i^k \in F \subset M$ de l'objet $o_k \in O$ telle que :*
$\forall o_k \in O$, $\exists\, méthodes_k \subset membres_k$,

$$\begin{aligned} Méthodes\colon O &\longrightarrow F \\ o_k &\longmapsto méthodes_k \equiv \{f_1^k, f_2^k, \cdots, f_j^k\} \end{aligned}$$

Ainsi, l'ensemble des membres d'un objet donné est représenté par l'union des champs et des méthodes de cet objet. Par exemple, les champs de l'objet `ip_packet` de la figure 3.6 ci-avant peuvent être calculés en listant les champs et méthodes de cet objet. C'est à dire :

$$O \equiv \{o_1, o_2, o_3, o_4, o_5, o_6, o_7, o_8, o_9, o_{10}, o_{11}, o_{12}, o_{13}, o_{14}\}$$

$$\begin{aligned} Noms(o_1) &\equiv \{\text{"}ip\_packet\text{"}\} \\ Types(o_1) &\equiv \{\text{"}ip\_header\_t\text{"}\} \\ Membres(o_1) &\equiv Champs(o_1) \cup Méthodes(o_1) \\ &\equiv \{a_1^1 = o_2, a_2^1 = o_3, \cdots, a_{13}^1 = o_{14}\} \cup \{\emptyset\} \end{aligned}$$

$$\begin{aligned} Noms(a_1^1) &\equiv \{\text{"}ip\_version\text{"}\}, \quad Types(a_1^1) \equiv \{\text{"}uint32\_t\text{"}\} \\ Membres(a_1^1) &\equiv Champs(a_1^1) \cup Méthodes(a_1^1) \equiv \emptyset \end{aligned}$$

$$Noms(a_2^1) \equiv \{\text{"}internet\_header\_length\text{"}\}, \quad Types(a_2^1) \equiv \{\text{"}uint32\_t\text{"}\}$$

$$Membres(a_2^1) \equiv Champs(a_2^1) \cup Méthodes(a_2^1) \equiv \emptyset$$
$$\vdots$$

Néanmoins, il faut remarquer que le calcul des membres d'un objet *méthode* diffère d'un objet traditionnel. En effet, une méthode représente du code exécutable, c'est à dire que les données d'une méthode correspondent aux instructions de la méthode. Or, du fait d'un résultat d'impossibilité sur l'analyse de programme [30], nous avons pour contrainte de ne pas nous intéresser à ce que réalise une méthode. Nous imposons donc qu'un objet *méthode* possède certes des données exécutables mais pas de membres. Une méthode est par conséquent un membre terminal.

**Propriété 3.1 (Membres d'une méthode)** *Soit $O$ un système à objets.*

$$\forall o_k \in O,\ \forall f_i^k \in Méthodes(o_k),\ Membres(f_i^k) = \emptyset$$

Il est à noter également qu'une méthode se distingue d'un champ par le fait qu'il s'agit d'un membre exécutable. C'est à dire qu'un objet *méthode* possède un type implicite qui est "exécutable". Certains systèmes comme Python [3] font d'ailleurs cette considération. De plus, une méthode se caractérise par l'ensemble des types des données qu'elle accepte en entrée ($IN$), en sortie ($OUT$) et en entrée/sortie ($INOUT$).

**Propriété 3.2 (Types d'une méthode)** *Soit $O$ un système à objets.*

$$\forall o_k \in O, \forall f_i^k \in Méthodes(o_k),\ \{"exécutable"\} \in Types(f_i^k)$$
$$\begin{aligned} Types(f_i^k) \equiv & \{"exécutable"\} \\ & \cup Types_{IN}(f_i^k) \\ & \cup Types_{OUT}(f_i^k) \\ & \cup Types_{INOUT}(f_i^k) \end{aligned}$$

*avec :*

$$\begin{aligned} Types_{IN}(f_i^k) &\equiv \{type_1^{IN}, \cdots, type_i^{IN}\} \\ Types_{OUT}(f_i^k) &\equiv \{type_1^{OUT}, \cdots, type_o^{OUT}\} \\ Types_{INOUT}(f_i^k) &\equiv \{type_1^{INOUT}, \cdots, type_{io}^{INOUT}\} \end{aligned}$$

3. https://docs.python.org/3.4/library/functions.html#callable

```
var lapin = {};
lapin.parler = function(tirade) {
  print("Le lapin dit '", tirade, "'");
};

lapin.parler("Eh bien, maintenant c'est vous qui me le demandez.");
```

FIGURE 3.7 – Exemple d'objet Javascript.

(crédits : `http://fr.eloquentjavascript.net/chapter8.html`

Prenons l'exemple du code Javascript de la figure 3.7. Celui-ci consiste à créer un objet *lapin* puis à le faire muter en lui ajoutant la méthode *parler*. Il faut savoir que Javascript est un système faiblement typé du fait du principe de mutation que nous expliquons dans la suite du document. En conséquence, cette méthode *parler* ne peut imposer de contraintes sur les types de son paramètre. De fait, elle accepte donc des objets dont le type n'est pas défini :

$$O \equiv \{o_1, o_2\}$$

$$\begin{aligned} Noms(o_1) &\equiv \text{"}lapin\text{"}, \quad Types(o_1) \equiv \text{"}objet\text{"} \\ Membres(o_1) &\equiv \{m_1^1 = o_2\} \end{aligned}$$

$$\begin{aligned} Noms(m_1^1) &\equiv \text{"}parler\text{"} \\ Types(m_1^1) &\equiv \{\text{"}exécutable\text{"}, type_1^{INOUT} = \text{"}non\text{-}défini\text{"}\} \\ Membres(m_1^1) &\equiv \emptyset \end{aligned}$$

### 3.1.8 Signature des objets

La signature est un élément d'information propre à chaque objet qui permet l'identification de ces objets. Contrairement aux noms, qui peuvent être arbitraires (cf. notation 3.2), la signature d'un objet se calcule à partir des éléments qui participent à son identification. En pratique, la signature d'un objet se dérive à partir de ses noms et de ses types.

**Notation 3.10 (Signature)** *Soit $O$ un environnement objets et $o_k$ un objet de $O$.*

*On définit l'application Signature qui pour un objet $o_k \in O$ renvoie tous les couples $(nom_i^k, type_j^k)$ tels que :*
$\forall o_k \in O, \exists\, signature_k \subset N \times T,$

$$\begin{aligned} Signature\colon O &\longrightarrow Noms(o_k) \times Types(o_k) \\ o_k &\longmapsto \quad signature_k \equiv noms_k \times types_k \end{aligned}$$

$$
\begin{aligned}
Signature(o_k) \equiv & Noms(o_k) \times Types(o_k) \\
\equiv & \{nom_1^k, nom_2^k, \cdots, nom_n^k\} \times \{type_1^k, type_2^k, \cdots, type_t^k\} \\
\equiv & \begin{Bmatrix}
(nom_1^k, type_1^k), & (nom_1^k, type_2^k), & \cdots, & (nom_1^k, type_t^k), \\
(nom_2^k, type_1^k), & (nom_2^k, type_2^k), & \cdots, & (nom_2^k, type_t^k), \\
\vdots & \vdots & \ddots & \vdots \\
(nom_n^k, type_1^k), & (nom_n^k, type_2^k), & \cdots, & (nom_n^k, type_t^k)
\end{Bmatrix}
\end{aligned}
$$

Plus concrètement, la signature permet surtout de lever les homonymies entre objets comme dans le cas de la surcharge de membres. Par exemple, la classe Java *Integer* déclare trois méthodes *valueOf* qui ne diffèrent que par les paramètres qu'elles acceptent en entrée. Ainsi, leurs noms n'est pas suffisant pour les distinguer mais leur signature respective si :

$$
\begin{aligned}
Noms(o_1) \equiv & "Integer" \\
Types(o_1) \equiv & "Classe" \\
Méthodes(o_1) \equiv & \{\cdots, f_5^1, f_6^1, f_7^1 \cdots\}
\end{aligned}
$$

$$
\begin{aligned}
Noms(f_5^1) \equiv & "valueOf", \quad Types(f_5^1) \equiv \{"exécutable", "Integer", "String", "entier"\} \\
Noms(f_6^1) \equiv & "valueOf", \quad Types(f_6^1) \equiv \{"exécutable", "Integer", "String"\} \\
Noms(f_7^1) \equiv & "valueOf", \quad Types(f_7^1) \equiv \{"exécutable", "Integer", "entier"\}
\end{aligned}
$$

$$
\begin{aligned}
Signature(f_5^1) \equiv & \{"valueOf"\} \times \{"exécutable", "Integer", "String", "entier"\} \\
Signature(f_6^1) \equiv & \{"valueOf"\} \times \{"exécutable", "Integer", "String"\} \\
Signature(f_7^1) \equiv & \{"valueOf"\} \times \{"exécutable", "Integer", "entier"\}
\end{aligned}
$$

On remarquera que les spécifications de la machine virtuelle Java [77] utilise une approche similaire mais propose d'encoder la signature avec une chaîne de caractères. Ainsi les signatures Java des méthodes de notre exemple sont respectivement :

$$
\begin{aligned}
Signature_{Java}(f_5^1) \equiv & "valueOf(Ljava/lang/String;I)Ljava/lang/Integer;" \\
Signature_{Java}(f_6^1) \equiv & "valueOf(Ljava/lang/String;)Ljava/lang/Integer;" \\
Signature_{Java}(f_7^1) \equiv & "valueOf(I)Ljava/lang/Integer;"
\end{aligned}
$$

Ainsi, la notion de signature est très utile pour lever les ambiguïtés dans la désignation des objets. Le système à objets lui même s'appuie dessus, généralement pour des raisons fonctionnelles. Pour nous, cette notion offre surtout la possibilité d'écrire des règles de contrôle applicables à des objets similaires.

En pratique néanmoins, la localisation sera généralement la seule information immédiatement accessible lors du contrôle. Parce que nous voulons déterminer les règles à appliquer à chaque objet, nous pouvons nous demander par exemple comment calculer la signature d'un objet C++ quand on ne dispose que d'un pointeur sur une suite d'octets. C'est pourquoi, nous introduisons une fonction qui associe à la localisation sa signature, c'est à dire les noms et types de l'objet à cet emplacement. Le contexte de sécurité de l'objet permet ansi de retrouver en pratique la signature et les règles à satisfaire pour celle-ci.

**Notation 3.11 (Contexte de sécurité)** *Soit $O$ un système à objets.*

*On définit l'application Contexte qui pour un objet $o_k \in O$ renvoie la signature de l'objet $o_k$ à partir de sa localisation telle que :*

$$\begin{aligned} \mathit{Contexte}: L &\longrightarrow N \times T \\ \mathit{Localisation}(o_k) &\longmapsto \quad \mathit{contexte}_k \equiv \mathit{noms}_k \times \mathit{types}_k \\ &\qquad\qquad\quad \equiv \mathit{Signature}(o_k) \end{aligned}$$

Au final, la notion de signature sert à définir des règles de contrôle et la notion de contexte est nécessaire pour retrouver les règles à satisfaire. Nous verrons dans le chapitre concernant l'implémentation, que la mise en œuvre de cette fonction repose sur un dialogue avec le système à objets considéré afin de retrouver la signature et les règles associées.

## 3.2 Systèmes à objets particuliers

La formalisation que nous avons proposée dans la section précédente est suffisante pour traiter un système à objets dans le cas général. Mais en pratique, chaque système dispose de ses propres particularités comme par exemple les systèmes à objets répartis où chaque objet peut être physiquement isolé des autres. C'est pourquoi, nous proposons dans cette section de formaliser les notions particulières des trois systèmes à objets que nous considérons : les langages à classes, les langages à prototypes et les systèmes à objets répartis (cf. tableau 3.1).

### 3.2.1 Systèmes basés sur les classes

Dans les environnements de programmation orientée objets il existe principalement deux stratégies pour construire les objets du système : l'approche basée sur

les classes et celle sur les prototypes. Nous détaillons l'approche par prototypes dans la sous-section suivante.

De façon générale, la notion de classe symbolise le "plan de fabrication" d'un type d'objet particulier. Ainsi, la classe doit être vue comme un objet qui définit l'ensemble des caractéristiques communes aux objets de même nature, c'est à dire la définition de leurs champs et méthodes. Cela implique alors que le classe représente le type d'un ou plusieurs objets ce qui revient à considérer que les types d'un objet sont égaux aux noms de sa classe. Nous verrons que la notion de classe est indissociable de la notion d'instance.

**Notation 3.12 (Classe)** *Soit $O$ un système à objets. Soit $C \subset O$ l'ensemble des classes connues de $O$.*

1. *On note $c_k \in C$ la classe d'un objet $o_k$ tel que :*
   $Types(o_k) \equiv Noms(c_k)$

2. *On définit l'application Classe qui pour un objet $o_k$ renvoie la classe $c_k$ de l'objet $o_k$ telle que :*
   $\forall o_k \in O,\ \exists! c_k \in C,$

$$\begin{aligned} Classe\colon O &\longrightarrow C \\ o_k &\longmapsto c_k \end{aligned}$$

```
1  class Chat:
2    count = 0
3
4    def __init__(self):
5      Chat.count += 1
6      self.id = Chat.count
7
8    def hello(self):
9      print ("I'm the %d th cat among %d" % (self.id, Chat.count, ))
```

FIGURE 3.8 – Exemple d'une classe en Python

Par exemple, le code python de la figure 3.8 déclare une classe *Chat* contenant un champ *count* ainsi que deux méthodes *__init__* et *hello*. C'est à dire qu'il s'agit d'un objet de type *Classe* nommé *Chat* et, selon notre notation, nous pouvons alors calculer :

$$C \equiv \{c_1\}, \quad O \equiv C \cup \{o_2, o_3, o_4\}$$

$$Noms(c_1) \equiv "Chat"$$

$$
\begin{aligned}
Types(c_1) &\equiv \text{"}Classe\text{"} \\
Champs(c_1) &\equiv \{a_1^1 = o_2\} \\
Méthodes(c_1) &\equiv \{f_1^1 = o_3, f_2^1 = o_4\}
\end{aligned}
$$

$$
\begin{aligned}
Signature(c_1 :: a_1) &\equiv (\text{"}count\text{"}, \text{"}entier\text{"}) \\
Membres(c_1 :: a_1) &\equiv \emptyset
\end{aligned}
$$

$$
\begin{aligned}
Signature(c_1 :: f_1) &\equiv \{(\text{"}\_init\_\text{"}, \text{"}exécutable\text{"}), (\text{"}\_init\_\text{"}, \text{"}non\text{-}défini\text{"})\} \\
Membres(c_1 :: f_1) &\equiv \emptyset
\end{aligned}
$$

$$
\begin{aligned}
Signature(c_1 :: f_2) &\equiv \{(\text{"}hello\text{"}, \text{"}exécutable\text{"}), (\text{"}hello\text{"}, \text{"}non\text{-}défini\text{"})\} \\
Membres(c_1 :: f_2) &\equiv \emptyset
\end{aligned}
$$

Dans un système à objets basé sur les classes, les objets sont construits à partir d'une classe par un processus appelé instanciation. Il s'agit de la composition de deux opérations, l'allocation et l'initialisation, qui sont représentées par deux méthodes particulières nommées respectivement "constructeur" et "initialisateur" (cf. section 3.4.2). Deux instances d'une même classe sont distinguables entre elles via un identifiant implicite appelé *numéro d'instance* qui est unique tout au long de la vie de l'objet.

Ainsi, chaque objet de ce type de système est obligatoirement soit une instance soit une classe ; la classe étant elle même l'instance d'une classe "Classe". Nous verrons dans la suite de ce chapitre ce que la notion de classe induit sur les relations entre objets.

**Notation 3.13 (Instances)** *Soit $O$ un système à objets.*

*On définit l'application Instance qui renvoie la liste des objets $o_i^k \in O$ instances d'une classe $c_k \in C$ telle que : $\forall c_k \in C$, $\exists instances_k \subset O$,*

$$
\begin{aligned}
Instances\colon C &\longrightarrow O \\
c_k &\longmapsto \ instances_k \equiv \{o_i \in O \mid Classe(o_i) \equiv c_k\}
\end{aligned}
$$

La figure 3.9 reprend l'exemple Python précédent mais en y ajoutant une méthode *main* (*if _name_ == "_main_" : ...*). La classe *Chat* déclare un champ *count* utilisé par la méthode *_init_* pour compter le nombre d'objets de type *Chat* instancié. Cette méthode, aussi appelée initialisateur, ajoute un champ *id*

```
1  class Chat:
2    count = 0
3
4    def __init__(self):
5      Chat.count += 1
6      self.id = Chat.count
7
8    def hello(self):
9      print ("I'm the %d th cat among %d" % (self.id, Chat.count, ))
10
11 if __name__ == "__main__":
12   chat1 = Chat()
13   chat2 = Chat()
14
15   chat1.sayHello()
16   chat2.sayHello()
```

FIGURE 3.9 – Exemple Python d'un champ partagé

pour chacun des objets *Chat* instancié dont la valeur est fixée à celle, incrémentée de 1, du champ *count* de la classe. À l'exécution de ce programme, deux instances de la classe *Chat* sont créées et pour lesquelles la valeur du champ *id* est respectivement fixé à 1 puis 2. En résumé, le champ *count* est commun à toutes les instances de la classe *Chat* et le champ *id* est particulier à chaque instance. Par rapport à l'exemple précédent, nous avons donc quatre objets supplémentaires qui correspondent aux deux instances créées par la méthode *main* et à leur champ *id* respectif. Ainsi selon notre notation, nous pouvons calculer :

$$C \equiv \{c_1\}, \quad O \equiv \{c_1, c_1 :: a_1, c_1 :: f_1, c_1 :: f_2, o_5, o_6, o_7, o_8\}$$

$$\begin{aligned} Instances(c_1) &\equiv \{o_5, o_6\} \\ Classe(o_5) &\equiv Classe(o_6) \equiv c_1 \end{aligned}$$

$$\begin{aligned} Noms(o_5) &\equiv \text{"}chat1\text{"} \\ Types(o_5) &\equiv Noms(Classe(o_5)) \equiv Noms(c_1) \equiv \text{"}Chat\text{"} \\ Champs(o_5) &\equiv \{a_1^5 = c_1 :: a_1, a_2^5 = o_7\} \\ Méthodes(o_5) &\equiv \{f_1^5 = c_1 :: f_1, f_2^5 = c_1 :: f_2\} \end{aligned}$$

$$\begin{aligned} Signature(o_5 :: a_2) &\equiv (\text{"}id\text{"}, \text{"}entier\text{"}) \\ Signature(o_5 :: f_1) &\equiv Signature(c_1 :: f_1) \\ &\equiv \{(\text{"}\_init\_\text{"}, \text{"}exécutable\text{"}), (\text{"}\_init\_\text{"}, \text{"}non\text{-}défini\text{"})\} \\ Signature(o_5 :: f_2) &\equiv Signature(c_1 :: f_2) \\ &\equiv \{(\text{"}hello\text{"}, \text{"}exécutable\text{"}), (\text{"}hello\text{"}, \text{"}non\text{-}défini\text{"})\} \end{aligned}$$

$$\begin{aligned} Noms(o_6) &\equiv \text{"}chat2\text{"} \\ Types(o_6) &\equiv Noms(Classe(o_6)) \equiv Noms(c_1) \equiv \text{"}Chat\text{"} \\ Champs(o_6) &\equiv \{a_1^6 = c_1 :: a_1, a_2^6 = o_8\} \\ Méthodes(o_6) &\equiv \{f_1^6 = c_1 :: f_1, f_2^6 = c_1 :: f_2\} \end{aligned}$$

$$
\begin{aligned}
Signature(o_6 :: a_2) \equiv & (\text{"}id\text{"}, \text{"}entier\text{"}) \\
Signature(o_6 :: f_1) \equiv & Signature(c_1 :: f_1) \\
\equiv & \{(\text{"}\_init\_\text{"}, \text{"}exécutable\text{"}), (\text{"}\_init\_\text{"}, \text{"}non\text{-}défini\text{"})\} \\
Signature(o_6 :: f_2) \equiv & Signature(c_1 :: f_2) \\
\equiv & \{(\text{"}hello\text{"}, \text{"}exécutable\text{"}), (\text{"}hello\text{"}, \text{"}non\text{-}défini\text{"})\}
\end{aligned}
$$

Au final, nous voyons que notre modèle général permet de définir les notions de classe et d'instance en utilisant simplement la notation proposée. Des contrôles spécifiques peuvent ainsi être proposés pour tous les objets d'une classe donnée ou pour une instance particulière par exemple.

### 3.2.2 Systèmes basés sur les prototypes

Un prototype est un objet modèle qui peut être cloné pour fabriquer de nouveaux objets. Contrairement aux classes qui fournissent une définition statique des objets, un prototype est par essence incomplet, non définitif et peut être amené à évoluer et s'enrichir en fonction des besoins. Ainsi, n'importe quel objet d'un environnement à prototypes peut s'auto-modifier ou modifier d'autres objets. Cela se traduit principalement par l'ajout, la suppression ou la modification de membres. Les nouveaux objets y sont alors créés par clonages et mutations successives.

**Notation 3.14 (Prototype)** *Soit $O$ un système à objets.*

1. *On note $p_k \in O$ le prototype d'un objet $o_k$ tel que :*
   $Types(p_k) \subset Types(o_k)$ *et* $Noms(p_k) \subset Noms(o_k)$

2. *On définit l'application Prototype qui pour un objet $o_k$ renvoie le prototype $p_k$ de l'objet $o_k$. C'est à dire :*
   $\forall o_k \in O, \exists! p_k \in O,$

$$
\begin{aligned}
Prototype\colon O &\longrightarrow O \\
o_k &\longmapsto p_k
\end{aligned}
$$

En pratique, les systèmes à prototypes doivent être vu comme une généralisation des systèmes basés sur les classes où chaque objet peut servir à construire de nouveaux objets. Par abus de langage, ces nouveaux objets ne sont alors

plus instances d'une classe mais instances d'un prototype. Nous verrons dans la suite de ce chapitre ce que la notion de prototype, et plus particulièrement la mutation, induit sur les relations entre objets.

Le langage de programmation Python est, par exemple, souvent présenté comme un langage à classes bien qu'en réalité ce soit un langage à prototypes. En effet, chaque objet est implémenté sous la forme d'un dictionnaire où chaque membre est indexé selon son nom [4] ce qui permet de modifier la définition de n'importe quel objet ; aspect caractéristique des langages à prototypes. Ainsi le programme Python de la figure 3.8 fait apparaître une mutation des instances de la classe *Chat* par la méthode *__init__* qui ajoute un champ *id* à chaque nouvelle instance. Selon notre notation, nous avons alors :

$$C \equiv \{c_1\}, \quad O \equiv \{c_1, c_1 :: a_1, c_1 :: f_1, c_1 :: f_2, o_5, o_6, o_5 :: a_2, o_6 :: a_2\}$$

$$\begin{aligned} Instances(c_1) &\equiv \{o_5, o_6\} \\ Classe(o_5) &\equiv Classe(o_6) \equiv c_1 \\ Prototype(o_5) &\equiv Prototype(o_6) \equiv c_1 \end{aligned}$$

Les capacités offertes par les langages à prototypes permettent à Python de simuler plusieurs paradigmes de programmation comme les langages à classes. L'exemple que nous avons choisi exploite cet aspect pour illustrer que notre notation s'applique sans difficultés aux langages à classes et aux langages à prototypes. Là encore, nous voyons que notre modèle général supporte directement et simplement les langages à prototypes et permet donc des règles de contrôle pour ces langages.

### 3.2.3 Systèmes à objets répartis

Lorsque l'on traite des systèmes à objets sans autre précision, nous faisons implicitement l'hypothèse que les objets du système résident tous dans une seule et même entité physique. Dans le cas d'un système à objets répartis, cette hypothèse n'est plus vraie car les objets d'un tel système ne résident plus sur une seule machine mais sont plutôt distribués/répartis sur plusieurs machines distinctes. Cette ségrégation physique des objets impose certaines propriétés à notre modèle général. En effet, si deux objets sont physiquement distants l'un de l'autre alors leur espace de localisation respectifs sont totalement disjoints.

4. voir la documentation Python sur les membres *__slots__* et *__dict__* (cf. `https://docs.python.org/3/reference/datamodel.html#objects-values-and-types`)

**Propriété 3.3 (Localisation d'objet répartis)** *Soit $O$ un système à objets répartis.*

1. *On définit $L \equiv \bigcup_{i=1}^{n} L_i$ l'espace des localisations d'objet possibles de $O$ tel que : $\forall L_i, L_j \subset L, L_i \cap L_j \equiv \emptyset$*

2. *Soit deux objets $o_1, o_2 \in O$ tels que : $Localisation(o_1) \in L_1 \subset L$ et $Localisation(o_2) \in L_2 \subset L$.*
   *Si $L_1 \neq L_2$ alors $o_1$ et $o_2$ sont deux objets distants. Si $L_1 \equiv L_2$ alors $o_1$ et $o_2$ sont deux objets centralisés.*

3. *Soit $L_k \subset L$, $\forall o_k \in O$, $Localisation(o_k) \in L_k \longrightarrow Localisation(Membres(o_k)) \in L_k$*

Ainsi, la localisation physique des objets du système induit des contraintes de visibilité sur les objets répartis. Dès lors, il convient d'affiner notre modèle général en distinguant trois situations : les systèmes centralisés, les systèmes parallèles et les systèmes répartis.

Un système centralisé s'assimile à un seul processus qui s'exécute sur une seule machine physique. Il s'agit du cas le plus simple car les objets résident dans un seul et même espace de localisation constitué d'un unique environnement physique. Dans une telle configuration les objets sont naturellement tous visibles les uns par rapport aux autres et il n'existe donc pas de limitations particulières pour qu'un objet ne soit pas accessible. Notre modèle général y reste donc valide.

Un système parallèle peut se voir comme plusieurs processus qui s'exécutent sur une seule et même machine. Les objets y sont distribués sur plusieurs espaces de localisation distincts mais appartenant tous au même environnement physique. Aussi, il suffit d'utiliser un mécanisme de type *mémoire partagée* pour qu'un objet distant soit visible par les autres objets du système ce qui permet d'exporter à la fois méthodes et champs d'objet. Néanmoins l'utilisation d'un tel mécanisme doit être explicite et par conséquent, les objets qui n'y ont pas recours, ne seront pas visibles par des objets distants. Par rapport à notre modèle général, cela signifie qu'un objet partagé appartient à plusieurs espaces de localisation. Notre modèle général reste donc là encore valide.

Un système réparti par contre doit être vu comme plusieurs processus qui s'exécutent sur plusieurs machines physiques. Les objets y sont distribués sur plusieurs espaces de localisation répartis sur plusieurs environnements physiques distincts reliés par un réseau. Dans une telle configuration il n'est donc plus

possible de faire appel à de la mémoire partagée pour rendre visible un objet distant. En conséquence, il n'est pas possible pour un objet d'accéder à l'espace d'un objet distant. La littérature sur les objets répartis contourne ce problème à l'aide d'un "contrat" qui liste les signatures des membres qu'un objet distant doit implémenter. Ce contrat est ensuite mis à disposition des autres objets pour que ceux-ci aient connaissance des membres disponibles pour cet objet distant. Il s'agit du concept d'interface (cf. figure 3.10). En pratique, une interface se traduit par un accord entre un *client* et un *serveur* où chacun d'entre eux implémente un objet de type *interface* en charge de sérialiser/désérialiser les appels de procédures distantes.

**Notation 3.15 (Interface)** *Soit $O$ un système à objets et $I$ l'ensemble des interfaces connues de $O$.*

1. *On définit l'application Interface qui pour un objet $o_k \in O$ renvoie une interface $i_k \in I$ correspondant à l'ensemble des signatures des membres $m_i^k \in M$ exportés par l'objet $o_k$ telle que :*
   $\forall o_k \in O, \exists i_k \subset I,$

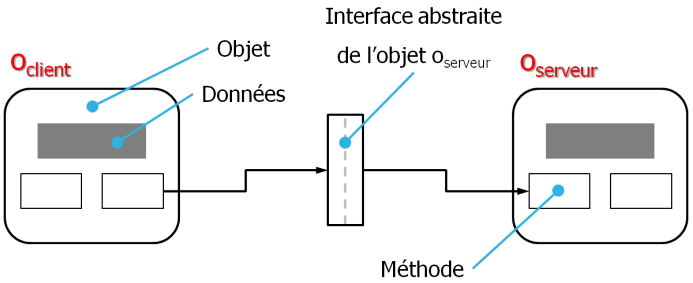
$$Interface : O \longrightarrow I$$
$$o_k \longmapsto \quad i_k \equiv \{Signature(m_i^k) | m_i^k \in Membres(o_k)\}$$

2. *Soit $L \equiv \bigcup_{i=1}^{n} L_i$ l'espace des localisation d'objet possibles de $O$.*
   $\forall o_k \in O, \forall L_i, L_j \subset L,$
   *Si $L_i \cap L_j \equiv \emptyset$ alors $Interface(o_k) \in L_i, L_j$*

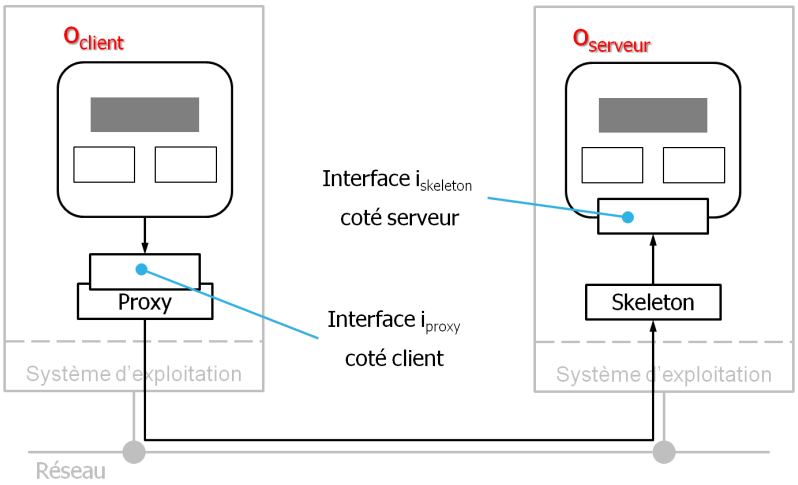
En pratique, les systèmes centralisés et les systèmes parallèles bénéficient des avantages d'un environnement physique commun ce qui autorise les interfaces à intégrer des signatures de champs comme c'est le cas avec Java par exemple. Mais dans le cas d'un système réparti il ne peut y avoir d'accès direct à la mémoire de l'objet distant et par conséquent les interfaces n'intègrent que des signatures de méthodes. Ainsi, notre modèle général couvre parfaitement ces trois familles de système pour peu que l'on tienne compte de la distance via la notion d'interface.

## 3.3 Modèle général de relations entre objets

Jusqu'à présent, nous nous sommes surtout intéressés aux éléments structurant des systèmes à objets. Or les objets sont par essence des entités qui s'échangent




(a) Représentation conceptuelle




(b) Mise en oeuvre

FIGURE 3.10 – Illustration du principe d'interface

$o_{client}$ et $o_{serveur}$ sont deux objets distants. Ils partagent un même contrat correspondant à l'interface abstraite de $o_{serveur}$. Celle-ci est projetée sous forme d'une interface concrète côté client et côté serveur (resp. $i_{proxy}$ et $i_{skeleton}$). L'interface cliente utilise un proxy applicatif pour transmettre les appels de procédure distante au squelette de l'interface serveur.

des données, communiquent et coopèrent pour accomplir les fonctions et fonctionnalités du système. Le but de cette section est donc de modéliser les différentes relations possibles entre objets. Ainsi nous pourrons évaluer et formaliser de façon précise ce qu'il est possible de contrôler.

**Hypothèse 3.1 (Relations contrôlable)** *Seules les relations réellement observables peuvent être contrôlées.*

La figure 3.11 présente la réflexion que nous avons suivie pour atteindre cet objectif. Nous sommes partis de l'idée que seules les relations observables entre objets pouvaient être formalisées et contrôlées. En pratique, nous pouvons observer un objet ayant accès à un autre objet (relation de référence), les opérations de lecture/écriture/exécution entre deux objets (relation d'interaction) et enfin les transferts d'informations et de contrôle entre les objets du système (relation de flux). En analysant les travaux antérieurs dont [78], nous proposons une logique de contrôle basée sur les relations observables. Nous faisons l'hypothèse que cette logique puisse être traduite en un algorithme implémentable sous la forme d'un automate de contrôle.

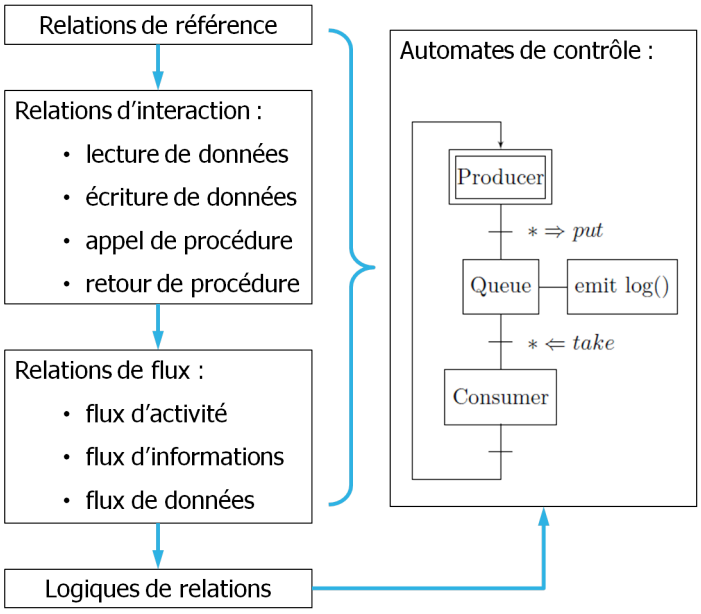



FIGURE 3.11 – Hiérarchisation des relations entre objets et organisation de cette section

Dans cette section nous ferons également des propositions pour contrôler les relations que nous formalisons. Dans les faits, l'utilité d'une politique nécessite de pouvoir observer ce que l'on veut contrôler. L'idée directrice est d'utiliser la même formalisation pour à la fois modéliser ce que l'on observe et pour exprimer les règles de contrôle. Pour cela, nous utiliserons d'un côté les contextes

de sécurité pour représenter les relations observées et de l'autre les signatures d'objet pour exprimer les politiques (cf. tableau 3.2).

| Relation observée | Règle exprimée |
|---|---|
| Relations de référence : | |
| $Contexte(loc_k) \dashrightarrow Contexte(loc_r)$ | $Signature(o_k) \dashrightarrow Signature(o_r)$ |
| Relations d'interaction : | |
| $Contexte(loc_k) \xrightarrow{read} Contexte(loc_r)$ | $Signature(o_k) \xrightarrow{read} Signature(o_r)$ |
| $Contexte(loc_k) \xrightarrow{write} Contexte(loc_r)$ | $Signature(o_k) \xrightarrow{write} Signature(o_r)$ |
| $Contexte(loc_k) \xrightarrow{invoke} Contexte(loc_r)$ | $Signature(o_k) \xrightarrow{invoke} Signature(o_r)$ |
| $Contexte(loc_k) \xrightarrow{return} Contexte(loc_r)$ | $Signature(o_k) \xrightarrow{return} Signature(o_r)$ |
| Relations de Flux : | |
| $Contexte(loc_k) \Longrightarrow Contexte(loc_r)$ | $Signature(o_k) \Longrightarrow Signature(o_r)$ |
| $Contexte(loc_k) \xRightarrow{data} Contexte(loc_r)$ | $Signature(o_k) \xRightarrow{data} Signature(o_r)$ |
| $Contexte(loc_k) \triangleright\triangleright Contexte(loc_r)$ | $Signature(o_k) \triangleright\triangleright Signature(o_r)$ |

Avec $loc_k \equiv Localisation(o_k)$ et $loc_r \equiv Localisation(o_r)$

TABLE 3.2 – Liens entre une relation et l'expression de sa règle de contrôle

### 3.3.1 Relations de référence

Pour accéder à un objet de l'environnement, il est nécessaire de connaître au préalable sa localisation. C'est à dire qu'un objet donné doit être capable de localiser de façon précise l'objet avec lequel il souhaite interagir. Cette connaissance, appelée référence, permet à cet objet d'accéder aux membres d'un autre objet ; soit en vue d'un appel de méthode, soit pour une lecture/écriture sur un champ. En pratique, une référence est une donnée de type primitif dont la valeur est égale à la localisation de l'objet référencé.

**Notation 3.16 (Références)** *Soit $O$ un système à objets et $o_k, o_r$ deux objets de $O$.*

1. *Soit $R$ l'ensemble des références de $O$. On définit l'application Références qui renvoie toutes les références que possède un objet $o_k \in O$ sur un objet $o_r \in O$ telle que :*
   $\forall o_k \in O, \exists! o_r \in O,$

   $$\textit{Références}: O \times O \longrightarrow R$$
   $$o_k, o_r \longmapsto \{ref_i^{k,r} \in R | Valeur(ref_i^{k,r}) \equiv Localisation(o_r)\}$$

2. *On note* $o_k \dashrightarrow o_r$ *le fait qu'un objet* $o_k$ *possède au moins une référence sur l'objet* $o_r$ *:*
   $o_k \dashrightarrow o_r \equiv \mathit{Références}(o_k, o_r) \neq \emptyset$

Dans un système informatique, un objet réside en mémoire et est utilisable par d'autres objets du même programme ou d'un programme distant lorsque l'on s'intéresse à des systèmes à objets répartis. En l'occurrence, la localisation de l'objet référencé peut être une adresse physique en mémoire, un indice dans un tableau, ou un triplet plateau/secteur/cluster lorsque l'on parle d'un emplacement sur des disques magnétiques, ou encore une référence réseau telle qu'une URL.

Par ailleurs, nous devons remarquer que lorsqu'un objet obtient une référence sur un autre objet du système, ce premier objet devient en capacité d'interagir avec l'objet référencé. La relation de référence ne signifie donc pas que deux objets vont effectivement interagir l'un avec l'autre mais plutôt que ceux-ci sont en mesure de le faire. Cela implique que l'existence d'une relation de référence entre deux objets ne présente pas un risque de sécurité immédiat. Néanmoins, l'obtention d'une référence peut ne pas être légitime car elle donne à l'objet référent le privilège implicite d'utiliser l'objet référencé. Ainsi et même si il n'y a pas d'accès effectif entre ces deux objets, l'objet référent obtient un privilège qu'il convient de contrôler.

C'est pourquoi, nous proposons de contrôler les relations de référence si le système considéré nous permet techniquement de remplir trois critères. C'est à dire, s'il est possible d'observer l'établissement de relations de référence ; d'identifier de façon unique les objets de cette relation ; et enfin s'il existe un moyen technique de bloquer cette mise en relation.

**Suggestion 3.1 (Contrôle des références)** *Soit* $O$ *un système à objets et* $o_k, o_r$ *deux objets de* $O$ *tels que* $\mathit{Localisation}(o_k) \equiv \mathit{loc}_k$ *et* $\mathit{Localisation}(o_r) \equiv \mathit{loc}_r$*.*

*La relation de référence* $o_k \dashrightarrow o_r$ *est contrôlable si :*

1. *La relation est techniquement observable.*

2. *Les entités Sujet et Objet (resp.* $o_k$ *et* $o_r$*) sont identifiables tel qu'il soit possible de calculer :*
   $o_k \dashrightarrow o_r \equiv \mathit{Contexte}(\mathit{loc}_k) \dashrightarrow \mathit{Contexte}(\mathit{loc}_r)$

3. *Il est possible d'appliquer une règle de contrôle de la forme :*
   $\mathit{deny/allow}\ \mathit{Signature}(o_k) \dashrightarrow \mathit{Signature}(o_r)$

Certains systèmes sont en mesure de limiter nativement l'obtention de références entre objets. En effet, la notion de référence est très liée à la notion de visibilité car si un objet n'est pas visible, il ne peut être référencé. Ainsi les contraintes physiques des systèmes répartis impliquent qu'aucun champ distant n'est référençable. Par ailleurs Java, comme d'autres environnements de programmation, propose un langage de politique basé sur les mots clefs *private/public* pour contrôler l'accès aux objets du système [5].

```
1  #include <stdio>
2
3  typedef struct {
4      unsigned short age;
5      bool a_des_nattes;
6    } fille_t;
7
8  static fille_t Pierette = { 15, false };
9
10 int main(int argc, char* argv[]) {
11
12   fille_t & ma_fille = Pierette;
13   fille_t & la_voisine = ma_fille;
14
15   la_voisine.a_des_nattes = true;
16   printf(ma_fille.a_des_nattes ? "oui\n" : "non\n");
17
18   return 0;
19 }
```

FIGURE 3.12 – Exemple d'utilisation des références en C++.

Ce programme C++ montre la création d'un objet de type *fille_t* appelé *Pierette*. Il s'en suit la déclaration d'une référence *ma_fille* sur cet objet, puis la déclaration d'une seconde référence *la_voisine*.

Supposons que nous soyons en mesure d'observer l'exécution du programme C++ de la figure 3.12. Lorsque les instructions correspondant aux lignes 12 et 13 sont exécutées, nous pouvons identifier que l'objet principal $o_7$ (*main*) obtient une référence sur un objet $o_4$ du système (*Pierette*) via deux variables de type référence. C'est à dire :

$$\begin{aligned} o_7 \dashrightarrow o_4 \equiv & Contexte(loc_7) \dashrightarrow Contexte(loc_4) \\ \equiv & Signature(o_7) \dashrightarrow Signature(o_4) \\ \equiv & Noms(o_7) \times Types(o_7) \dashrightarrow Noms(o_4) \times Types(o_4) \\ \equiv & (\text{"}main\text{"}, \text{"}exécutable\text{"}) \dashrightarrow (\text{"}Pierette\text{"}, \text{"}fille\_t\text{"}) \end{aligned}$$

Avec : $loc_7 \equiv Localisation(o_7)$ et $loc_4 \equiv Localisation(o_4)$

Ainsi nous pouvons modéliser cette relation de référence au moment de son établissement. Il ne nous reste donc plus qu'à comparer cette relation avec une politique de sécurité afin de trouver une règle de la forme *deny/allow* ("*main*", "*exécutable*") $\dashrightarrow$ ("*Pierette*", "*fille_t*").

5. http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

### 3.3.2 Relations d'interaction

Dans un système à objets, chaque objet coopère avec les autres afin de réaliser les fonctions et fonctionnalités du système. Le comportement global du système dépend alors des interactions entre les objets qui le composent. Ces interactions ont alors pour effet de produire une modification de l'état des objets qui interagissent. C'est à dire soit un échange de données entre objets, soit le passage de contrôle du système d'un objet vers l'autre, soit les deux. Plus particulièrement, les interactions possibles entre deux objets correspondent aux opérations élémentaires que sont la lecture, l'écriture et l'exécution.

#### Types d'interactions entre objets

Imaginons une situation de la vie courante où une personne saisit un stylo posé sur une table. Nous pouvons aisément admettre que cette personne décide d'elle même de saisir ce stylo mais qu'elle a besoin de sa main pour réaliser l'action correspondante. Les objets agissent exactement de la même manière, c'est à dire que chaque objet du système interagit avec les autres par l'intermédiaire de ses membres. Bien entendu, nous faisons là une hypothèse anthropomorphique mais, selon nous, peu importe si les objets répondent à une logique de programmation ou non, ce qui est important c'est ce qu'ils font.

**Hypothèse 3.2 (Aspect anthropomorphique des objets)** *Que ce soit par programmation, intelligence ou autre, un objet réalise des actions et il utilise ses membres pour les mettre en application.*

Ainsi, si les opérations de lecture/écriture correspondent effectivement à un transfert de données entre deux objets, l'exécution est quant à elle la composition de deux opérations que sont l'appel et le retour de procédure. Il s'agit respectivement du transfert du contrôle du programme de l'appelant vers l'appelé puis de l'appelé vers l'appelant. C'est pourquoi nous avons besoin de définir l'ensemble des objets qui possèdent le contrôle du système et dont la particularité est qu'il contient autant d'objets qu'il y a d'activités dans le système.

**Notation 3.17 (Lecture)** *Soit $O$ un système à objets et $o_k, o_r$ deux objets de $O$.*

1. *On définit la fonction Lecture qui copie les valeurs d'un membre $m_j^r$ de l'objet $o_r \in O$ vers un membre $m_i^k$ de l'objet $o_k \in O$ telle que :*
   $\forall o^k, o^r \in O, \forall m_i^k, m_j^r \in M, \exists ref^{k,r} \in Références(o_k, o_r)$,

$$\begin{aligned} Lecture\colon (O \times M)^2 &\longrightarrow V \\ (o_k, m_i^k, o_r, m_j^r) &\longmapsto \{Valeur(m_i^k) = Valeur(m_j^r)\} \end{aligned}$$

2. *On note $o_k \xrightarrow{m_i^k \ read \ m_j^r} o_r$ une opération de lecture initiée par l'objet $o_k$ sur l'objet $o_r$ telle que :*
   $o_k \xrightarrow{m_i^k \ read \ m_j^r} o_r \equiv Lecture(o_k, m_i^k, o_r, m_j^r)$

**Notation 3.18 (Appel)** *Soit $O$ un système à objets et $o_k, o_r$ deux objets de $O$.*

1. *Soit l'ensemble $S \subset O$ des objets qui possèdent le contrôle du système. On note $\overset{\bullet}{o_k}$ un objet $o_k$ de cet ensemble.*

2. *On définit la fonction Appel qui passe le contrôle du système d'un objet $o_k \in O$ à un objet $o_r \in O$ lorsque le membre $m_i^k$ de $o_k$ appel/invoque le membre $m_j^r$ de $o_r$ telle que :*
   $\forall o^k, o^r \in O, \forall m_i^k, m_j^r \in M, \exists ref^{k,r} \in Références(o_k, o_r)$,

$$\begin{aligned} Appel\colon (S \times M) \times (O \times M) &\longrightarrow O \times S \\ (\overset{\bullet}{o_k}, m_i^k, o_r, m_j^r) &\longmapsto (o_k, \overset{\bullet}{o_r}) \end{aligned}$$

3. *On note $o_k \xrightarrow{m_i^k \ invoke \ m_j^r} o_r$ une opération d'appel initiée par l'objet $o_k$ sur l'objet $o_r$ telle que :*
   $o_k \xrightarrow{m_i^k \ invoke \ m_j^r} o_r \equiv Appel(\overset{\bullet}{o_k}, m_i^k, o_r, m_j^r)$

**Notation 3.19 (Écriture)** *Soit $O$ un système à objets et $o_k, o_r$ deux objets de $O$.*

1. *On définit la fonction Écriture qui copie les valeurs d'un membre $m_i^k$ de l'objet $o_k \in O$ vers un membre $m_j^r$ de l'objet $o_r \in O$ telle que :*
   $\forall o^k, o^r \in O, \forall m_i^k, m_j^r \in M, \exists ref^{k,r} \in Références(o_k, o_r)$,

$$\begin{aligned} Écriture\colon (O \times M)^2 &\longrightarrow V \\ (o_k, m_i^k, o_r, m_j^r) &\longmapsto \{Valeur(m_j^r) = Valeur(m_i^k)\} \end{aligned}$$

2. *On note $o_k \xrightarrow{m_i^k \ write \ m_j^r} o_r$ une opération d'écriture initiée par l'objet $o_k$ sur l'objet $o_r$ telle que :*
   $o_k \xrightarrow{m_i^k \ write \ m_j^r} o_r \equiv Écriture(o_k, m_i^k, o_r, m_j^r)$

**Notation 3.20 (Retour)** *Soit $O$ un système à objets et $o_k, o_r$ deux objets de $O$.*

1. *Soit l'ensemble $S \subset O$ des objets qui possèdent le contrôle du système. On note $\overset{\bullet}{o_k}$ un objet $o_k$ de cet ensemble.*

2. *On définit la fonction Retour qui passe le contrôle du système d'un objet $o_r \in O$ à un objet $o_k \in O$ lorsque le membre $m_j^r$ de $o_r$ met fin à son exécution pour retourner au membre d'appel $m_i^k$ de $o_k$ telle que :*
   $\forall o^k, o^r \in O, \forall m_i^k, m_j^r \in M, \exists ref^{k,r} \in Références(o_k, o_r)$,

$$\begin{aligned} Retour\colon (O \times M) \times (S \times M) &\longrightarrow S \times O \\ (o_k, m_i^k, \overset{\bullet}{o_r}, m_j^r) &\longmapsto (\overset{\bullet}{o_k}, o_r) \end{aligned}$$

3. *On note $o_r \xrightarrow{m_j^r \ return \ m_i^k} o_k$ une opération de retour initiée par l'objet $o_r$ sur l'objet $o_k$ telle que :*
   $o_r \xrightarrow{m_j^r \ return \ m_i^k} o_k \equiv Retour(o_k, m_i^k, \overset{\bullet}{o_r}, m_j^r)$

**Discussions**

Contrôler les interactions entre objets est possible si et seulement si le système le permet techniquement. C'est à dire que, comme pour les relations de référence, nous devons être en mesure d'observer ces interactions et d'y appliquer des règles de contrôle d'accès.

**Suggestion 3.2 (Contrôle des interactions)** *Soit $O$ un système à objets et $o_k, o_r$ deux objets de $O$ tels que $Localisation(o_k) \equiv loc_k$ et $Localisation(o_r) \equiv loc_r$.*

*Toute interaction $o_k \longrightarrow o_r$ est contrôlable si :*

1. *La relation est techniquement observable.*
2. *Les entités Sujet et Objet (resp. $o_k$ et $o_r$) sont identifiables telles qu'il soit possible de calculer :*
   *$o_k \longrightarrow o_r \equiv Contexte(loc_k) \longrightarrow Contexte(loc_r)$*
3. *Les membres source et cible (resp. $m_i^k$ et $m_j^r$) sont identifiables tel qu'il soit possible de calculer :*
   *$Signature(m_i^k)$ et $Signature(m_j^r)$*
4. *Il est possible d'appliquer une règle de contrôle de la forme :*
   *$deny/allow\ Signature(o_k) \longrightarrow Signature(o_r)$*

*Ces quatre points se factorisent de manière à formaliser l'interaction observée et/ou la règle de contrôle exprimée entre deux objets du système comme par exemple :*

$$Contexte(loc_k) \xrightarrow{Signature(m_i^k)\ invoke\ Signature(m_j^r)} Contexte(loc_r)$$

Cependant, nous avions établi dans la section précédente qu'un membre pouvait être soit un champ, soit une méthode, ce qui implique que nous devons distinguer quatre grands cas d'interaction entre objets :

— Les interactions *Méthode/Méthode* ($f_i^k \longrightarrow f_j^r$) correspondent le plus souvent à une méthode qui en exécute une autre. C'est à dire la composition d'un appel de méthode suivi d'un retour. Les opérations de lecture/écriture sont également possibles entre méthodes sous réserve qu'au moins une des deux mute en un champ. Ainsi, l'interaction $f_i^k \xrightarrow{read} f_j^r$ est possible si la méthode $f_i^k$ mute implicitement en un champ $a_i^k$, par exemple.

— Les interactions *Méthode/Champ* ($f_i^k \longrightarrow a_j^r$) sont également un cas trivial puisqu'il s'agit le plus souvent d'une opération de lecture/écriture sur des données. L'exécution d'un champ est également possible mais fait que le champ mute implicitement en une méthode. Ainsi, l'interaction $f_i^k \xrightarrow{invoke} a_j^r$ est possible si le champ $a_j^r$ mute implicitement en une méthode $f_j^r$, par exemple.

— Les interactions *Champ/Champ* ($a_i^k \longrightarrow a_j^r$) correspondent généralement à une affectation entre deux champs. Indirectement, une opération de lecture suivie d'une écriture via une même méthode correspond aussi à une affectation mais qui est à rapprocher du cas précédent comme, par exemple, une interaction $f_p^z \xrightarrow{read} a_j^r$ suivie de $f_p^z \xrightarrow{write} a_i^k$ [79]. Les opérations d'exécution sont possibles entre deux champs sous réserve que ces deux champs mutent en deux méthodes ce qui est à rapprocher du premier cas considéré. Par exemple, l'interaction $a_i^k \xrightarrow{invoke} a_j^r$ est possible si le champ $a_i^k$ (resp. $a_j^r$) mute implicitement en une méthode $f_i^k$ (resp. $f_j^r$).

— Les interactions *Champ/Méthode* ($a_i^k \longrightarrow f_j^r$) sont théoriquement possibles dans notre modèle mais nous n'avons pas pu identifier de cas concret qui illustre ce type d'interaction. Plus vraisemblablement, nous pensons que ce cas force la mutation des membres concernés pour correspondre à l'un des trois cas précédents.

Or, bien que notre modèle supporte quatre cas d'interactions possibles, la plupart du temps nous observerons surtout des interactions *Méthode/Méthode* ou *Méthode/Champ*. Par ailleurs, beaucoup de systèmes à objets tels que Java considèrent que le fait d'accéder directement à un champ est une mauvaise pratique et favorise en conséquence l'utilisation de méthodes du genre accesseur/mutateur. Au final, notre modèle traitera donc principalement les interactions de type *Méthode/Méthode*.

### Cas d'étude : Appel de procédure distante

Illustrons ce type de relation à l'aide du code python de la figure 3.13. Ce programme réalise un appel de procédure distante d'un objet $o_{client}$ sur un objet $o_{serveur}$ via une interface XML-RPC [6]. Cette interface induit la création d'un objet $i_{proxy}$ côté client qui sérialise les appels reçus et d'un objet $i_{skeleton}$ côté serveur qui transmet les appels reçus par $i_{proxy}$ à l'objet $o_{serveur}$.

Supposons que nous ne soyons pas en mesure d'analyser le réseau entre les machines qui exécuteront respectivement ces deux programmes. Nous pouvons observer l'appel de procédure distante directement depuis ces deux systèmes au

6. C'est à dire l'utilisation d'XML pour la sérialisation des données et HTTP pour le transport.

```python
from xmlrpc.server import SimpleXMLRPCServer

def is_even(n):
  return n%2 == 0

if __name__ == "__main__":

  # Registering skeleton interface
  server = SimpleXMLRPCServer(("localhost", 8000))
  server.register_function(is_even, "is_even")
  print("Listening on port 8000...")

  # Waiting for remote procedure calls
  server.serve_forever()
```

(a) Serveur XML-RPC

```python
from xmlrpc.client import ServerProxy

if __name__ == "__main__":

  # Registering proxy interface
  proxy = ServerProxy("http://localhost:8000/")

  # Making remote procedure calls
  print("3 is even: %s" % str(proxy.is_even(3)))
  print("100 is even: %s" % str(proxy.is_even(100)))
```

(b) Client XML-RPC

FIGURE 3.13 – Exemple d'appel de procédures distantes en Python

(crédits : `https://docs.python.org/3/library/xmlrpc.client.html`)

moment de l'exécution des lignes 9 et 10 du client XML-RPC. C'est à dire que nous observerons côté client :

$$f_1^{client} \xrightarrow{invoke} f_2^{serveur}$$
$$\equiv o_{client} \xrightarrow{f_1^{client}\ invoke\ f_2^{serveur}} i_{proxy}$$
$$\equiv Contexte(loc_{client}) \xrightarrow{Signature(f_1^{client})\ invoke\ Signature(f_2^{serveur})} Contexte(loc_{proxy})$$
$$\equiv Signature(o_{client}) \xrightarrow{Signature(f_1^{client})\ invoke\ Signature(f_2^{serveur})} Signature(i_{proxy})$$
$$\equiv \text{"}Client\ XML-RPC\text{"} \xrightarrow{\text{"}main\text{"}\ invoke\ \text{"}is\_even\text{"}} \text{"}proxy\text{"}$$

Pour satisfaire au principe d'interface, toute interaction reçue par l'objet $i_{proxy}$ est transmise à l'objet $i_{skeleton}$. En conséquence, nous observerons côté serveur :

$$f_1^{client} \xrightarrow{invoke} f_2^{serveur}$$
$$\equiv i_{skeleton} \xrightarrow{f_1^{client}\ invoke\ f_2^{serveur}} o_{serveur}$$
$$\equiv Contexte(loc_{skeleton}) \xrightarrow{Signature(f_1^{client})\ invoke\ Signature(f_2^{serveur})} Contexte(loc_{serveur})$$
$$\equiv Signature(i_{skeleton}) \xrightarrow{Signature(f_1^{client})\ invoke\ Signature(f_2^{serveur})} Signature(o_{serveur})$$
$$\equiv \text{"}server\text{"} \xrightarrow{\text{"}main\text{"}\ invoke\ \text{"}is\_even\text{"}} \text{"}Serveur\ XML-RPC\text{"}$$

Enfin, lorsque l'objet $o_{serveur}$ termine l'exécution de la méthode *is_even*, nous pouvons observer les retours successifs à l'objet $o_{client}$, d'abord côté serveur puis côté client :

$$f_2^{serveur} \xrightarrow{return} f_1^{client}$$
$$\equiv o_{serveur} \xrightarrow{f_2^{serveur}\ return\ f_1^{client}} i_{skeleton}$$
$$\equiv Contexte(loc_{serveur}) \xrightarrow{Signature(f_2^{serveur})\ return\ Signature(f_1^{client})} Contexte(loc_{skeleton})$$
$$\equiv Signature(o_{serveur}) \xrightarrow{Signature(f_2^{serveur})\ return\ Signature(f_1^{client})} Signature(i_{skeleton})$$
$$\equiv \text{"}Serveur\ XML-RPC\text{"} \xrightarrow{\text{"}is\_even\text{"}\ return\ \text{"}main\text{"}} \text{"}server\text{"}$$

$$f_2^{serveur} \xrightarrow{return} f_1^{client}$$
$$\equiv i_{proxy} \xrightarrow{f_2^{serveur}\ return\ f_1^{client}} o_{client}$$
$$\equiv Contexte(loc_{proxy}) \xrightarrow{Signature(f_2^{serveur})\ return\ Signature(f_1^{client})} Contexte(loc_{client})$$
$$\equiv Signature(i_{proxy}) \xrightarrow{Signature(f_2^{serveur})\ return\ Signature(f_1^{client})} Signature(o_{client})$$
$$\equiv \text{"}proxy\text{"} \xrightarrow{\text{"}is\_even\text{"}\ return\ \text{"}main\text{"}} \text{"}Client\ XML-RPC\text{"}$$

Ainsi, nous voyons que nous sommes en mesure de représenter les interactions entre objets et grâce à cela nous pouvons appliquer des règles pour contrôler le système à objets considéré. Mais plus encore, cet exemple montre aussi que dans le cas d'objets distants, la représentation d'une même interaction est possible entre deux systèmes distants. Par exemple, le cas d'étude donné par [80] utilise notre modèle de relation pour suivre une interaction entre deux applications Android. Ainsi, nous pouvons appliquer pour une même interaction soit des règles de contrôle communes aux deux systèmes ou bien appliquer des règles différentes en fonctions des objectifs de sécurité.

### 3.3.3 Relations de flux

Quel que soit le système considéré, la notion de flux correspond à une relation abstraite qui s'établit entre les objets en fonction de la nature de leurs interactions. On distingue généralement trois types de flux : le flux de contrôle, le flux d'information et le flux de données. De façon générale, les flux d'un système à objets ne sont pas nécessairement légitimes/voulus et c'est pourquoi nous nous proposons de les contrôler. Il serait dommageable, par exemple, qu'un objet du système fasse fuiter de l'information confidentielle via un flux caché et/ou non contrôlé.

#### Flux d'activité

Nous pouvons d'abord remarquer que lorsque deux objets interagissent, ceux-ci s'échangent implicitement le contrôle du système pour mettre en œuvre les opérations demandées. Ainsi, un objet appelé conserve le contrôle du système jusqu'à ce que celui-ci opère une opération de retour vers l'objet ayant initié l'appel. De fait, le contrôle du système doit être effectué à l'échelle de chaque objet en fonction des différentes interactions et non de façon globale (c'est à dire à l'échelle du thread ou du processus). Afin d'éviter toute confusion entre les termes nous préférerons l'expression *flux d'activité* pour qualifier ce type de flux.

**Notation 3.21 (Flux d'activité)** *Soit $O$ un système à objets et $o_k, o_r$ deux objets de $O$.*

*On note $o_k \triangleright\triangleright o_r$ le transfert de contrôle du système de l'objet $o_k$ vers l'objet $o_r$ tel que :*

$$o_k \triangleright\triangleright o_r \equiv \begin{cases} Appel(o_k, m_i^k, o_r, m_j^r) \\ ou \\ Retour(o_k, m_i^k, o_r, m_j^r) \end{cases}$$

Prenons l'exemple d'une méthode $o_2 :: f_1$ qui calcule la somme de deux entiers. Lorsque cette méthode sera exécutée par une méthode $o_1 :: f_1$, cette dernière arrête momentanément son exécution pour donner la main à la méthode appelée et ainsi réaliser le calcul demandé. Puis lorsque la méthode $o_2 :: f_1$ retourne une fois le calcul effectué, la méthode $o_1 :: f_1$ reprend son exécution. Nous pouvons ainsi observer un flux d'activité d'abord de l'objet $o_1$ vers l'objet $o_2$ puis de $o_2$ vers $o_1$ :

$$o_1 \xrightarrow{f_1^1 \ invoke \ f_1^2} o_2 \text{ implique } o_1 \triangleright\triangleright o_2$$

$$o_2 \xrightarrow{f_1^2 \ return \ f_1^1} o_1 \text{ implique } o_2 \triangleright\triangleright o_1$$

**Flux d'information**

Bien que deux objets peuvent interagir sans s'échanger de données, certaines interactions induisent un transfert effectif d'information entre objets. Lorsque c'est le cas, une relation abstraite s'établit où au moins l'un des deux objets de l'interaction est source d'information pour l'autre. C'est ce que la littérature appelle flux d'information. En pratique, un flux d'information est indépendant du sens de l'interaction et de fait il peut être soit dans un sens opposé à l'interaction, soit dans le même sens, soit dans les deux sens. En effet, une opération de lecture par exemple ($o_k \xrightarrow{read} o_r$) induit un transfert de données de l'objet lu ($o_r$) vers l'objet lecteur ($o_k$). De même, une opération d'écriture ($o_k \xrightarrow{write} o_r$) induit un transfert de données de l'objet ayant initié cette opération ($o_k$) vers celui qui la reçoit ($o_r$). Enfin, l'exécution d'une méthode ($o_k \xrightarrow{invoke} o_r$) peut quant à elle induire un transfert d'information dans les deux sens (d'abord de $o_k$ vers $o_r$ puis de $o_r$ vers $o_k$).

**Notation 3.22 (Flux d'informations)** *Soit $O$ un système à objets et $o_k, o_r$ deux objets de $O$.*

1. *On note $o_k \Longrightarrow o_r$ un flux d'informations de l'objet $o_k$ vers l'objet $o_r$ tel que :*
   $\forall o_k, o_r \in O, \forall m_i^k, m_j^r \in M, \exists o_k \longrightarrow o_r,$
   $$o_k \Longrightarrow o_r \equiv \begin{cases} \acute{E}criture(o_k, m_i^k, o_r, m_j^r) \neq \emptyset \\ ou \\ Appel(o_k, m_i^k, o_r, m_j^r) \neq \emptyset \end{cases}$$
2. *On note $o_k \Longleftarrow o_r$ un flux d'informations de l'objet $o_r$ vers l'objet $o_k$ tel que :*

$$\forall o_k, o_r \in O, \forall m_i^k, m_j^r \in M, \exists o_k \longrightarrow o_r,$$
$$o_k \Longleftarrow o_r \equiv \begin{cases} Lecture(o_k, m_i^k, o_r, m_j^r) \neq \emptyset \\ ou \\ Retour(o_k, m_i^k, o_r, m_j^r) \neq \emptyset \end{cases}$$

*Remarque : Nous utilisons ici une définition plus large que celle de [78] car cette dernière se limite aux seuls objets du système d'exploitation.*

Prenons l'exemple de la méthode qui calcule la somme de deux entiers. Lorsque la méthode $o_1 :: f_1$ appelle la méthode $o_2 :: f_1$, nous pouvons observer un flux d'information bidirectionnel entre les objets $o_1$ et $o_2$. D'abord un flux de l'objet $o_1$ vers l'objet $o_2$ pour le passage des paramètres puis un flux d'information de l'objet $o_2$ vers $o_1$ correspondant au résultat de l'addition des deux entiers. C'est à dire :

$$o_1 \xrightarrow{f_1^1 \ invoke \ f_1^2} o_2 \text{ induit } o_1 \Longrightarrow o_2 \text{ car } Types_{IN}(f_1^2) \equiv \{\text{"}entier\text{"}, \text{"}entier\text{"}\}$$

$$o_2 \xrightarrow{f_1^2 \ return \ f_1^1} o_1 \text{ induit } o_2 \Longrightarrow o_1 \text{ car } Types_{OUT}(f_1^2) \equiv \{\text{"}entier\text{"}\}$$

**Flux de données**

Ainsi, si nous sommes en mesure de détecter certaines interactions entre deux objets alors nous pouvons contrôler les flux d'information entre ces deux objets. Par contre, la limite de cette relation est qu'elle ne tient pas compte de la valeur des données échangées ce qui entraîne une approximation sur la légitimité des données du flux. Par exemple, il peut être légitime d'autoriser un flux d'information entre deux objets sauf lorsque les données échangées sont de nature confidentielle. C'est pourquoi nous nous proposons de nous intéresser également à la valeur des données transférées de sorte à lever toute ambiguïté sur la nature des informations lorsque cela s'avère nécessaire.

**Notation 3.23 (Flux de données/valeurs)** *Soit $O$ un système à objets et $o_k, o_r$ deux objets de $O$.*

1. *On note $o_k \overset{data}{\Longrightarrow} o_r$ l'écriture d'une donnée data de l'objet $o_k$ vers l'objet $o_r$ telle que :*
   $\forall o_k, o_r \in O, \forall m_i^k, m_j^r \in M, \exists o_k \Longrightarrow o_r,$
   $$o_k \overset{data}{\Longrightarrow} o_r \equiv \begin{cases} data = \acute{E}criture(o_k, m_i^k, o_r, m_j^r) \\ ou \\ data = Appel(o_k, m_i^k, o_r, m_j^r) \end{cases}$$
2. *On note $o_k \overset{data}{\Longleftarrow} o_r$ la lecture d'une donnée data de l'objet $o_r$ vers l'objet $o_k$ telle que :*
   $\forall o_k, o_r \in O, \forall m_i^k, m_j^r \in M, \exists o_k \Longleftarrow o_r,$
   $$o_k \overset{data}{\Longleftarrow} o_r \equiv \begin{cases} data = Lecture(o_k, m_i^k, o_r, m_j^r) \\ ou \\ data = Retour(o_k, m_i^k, o_r, m_j^r) \end{cases}$$

Ainsi, continuons avec notre exemple sur la somme de deux entiers. En analysant la valeur des paramètres au moment de l'appel puis la valeur retournée à la fin de l'exécution, nous pouvons observer les données échangées entre les objets $o_1$ et $o_2$. C'est à dire que si la méthode $o_2 :: f_1$ est appelée pour calculer la somme du nombre 29 avec le nombre 13 alors nous observons :

$$o_1 \xrightarrow{f_1^1 \; invoke \; f_1^2} o_2 \text{ porte le flux de données } o_1 \overset{\{29,13\}}{\Longrightarrow} o_2$$

$$o_2 \xrightarrow{f_1^2 \; return \; f_1^1} o_1 \text{ porte le flux de données } o_2 \overset{\{42\}}{\Longrightarrow} o_1$$

**Discussions**

En pratique, le contrôle des flux du système offre une meilleure précision que celui des interactions entre objets. Par exemple, en utilisant les flux de données, nous pouvons déterminer la légitimité d'un appel de méthode en fonction de la valeur des paramètres ce qui évite d'avoir à bloquer tout appel à cette méthode. Par contre, le contrôle des flux induit une étape de formalisation supplémentaire qui aura forcément un impact sur les performances. Nous pensons cependant que les bénéfices sont suffisamment intéressants pour justifier la pertinence de ce type de contrôle.

**Suggestion 3.3 (Contrôle des flux d'activité)** *Soit $O$ un système à objets et $o_k, o_r$ deux objets de $O$.*

*Tout flux de contrôle $o_k \triangleright\triangleright o_r$ est contrôlable si :*

1. *Le flux est techniquement observable.*
2. *Les entités Sujet et Objet sont identifiables tel que :*
   $o_k \triangleright\triangleright o_r \equiv Contexte(o_k) \triangleright\triangleright Contexte(o_r)$
3. *Il est possible d'appliquer une règle de contrôle de la forme :*
   *deny* $Signature(o_k) \triangleright\triangleright Signature(o_r)$

**Suggestion 3.4 (Contrôle des flux d'information)** *Soit $O$ un système à objets et $o_k, o_r$ deux objets de $O$.*

*Tout flux d'information $o_k \Longrightarrow o_r$ est contrôlable si :*

1. *Le flux est techniquement observable.*
2. *Les entités Sujet et Objet sont identifiables tel que :*
   $o_k \Longrightarrow o_r \equiv Contexte(o_k) \Longrightarrow Contexte(o_r)$
3. *Il est possible d'appliquer une règle de contrôle de la forme :*
   *deny* $Signature(o_k) \Longrightarrow Signature(o_r)$

**Suggestion 3.5 (Contrôle des flux de données/valeurs)** *Soit $O$ un système à objets et $o_k, o_r$ deux objets de $O$.*

*Tout flux de données $o_k \overset{data}{\Longrightarrow} o_r$ est contrôlable si :*

1. *Le flux est techniquement observable.*
2. *Les entités Sujet et Objet sont identifiables tel que :*
   $o_k \overset{data}{\Longrightarrow} o_r \equiv Contexte(o_k) \overset{data}{\Longrightarrow} Contexte(o_r)$
3. *Il est possible d'appliquer une règle de contrôle de la forme :*
   *deny* $Signature(o_k) \overset{data}{\Longrightarrow} Signature(o_r)$

**Cas d'étude : Modèle producteur/consommateur**

Illustrons ce type de relation en prenant l'exemple du modèle producteur/consommateur. Il s'agit d'un patron de conception qui permet à plusieurs objets de se transmettre de l'information de façon asynchrone par l'intermédiaire d'un objet tampon partagé. En pratique il se traduit par la mise en œuvre d'au moins trois objets : celui qui produit la donnée, celui qui la reçoit et celui qui en assure la transmission. La figure 3.14 montre le résultat de l'exécution du programme Java de la figure 3.15 qui implémente ce patron de conception.

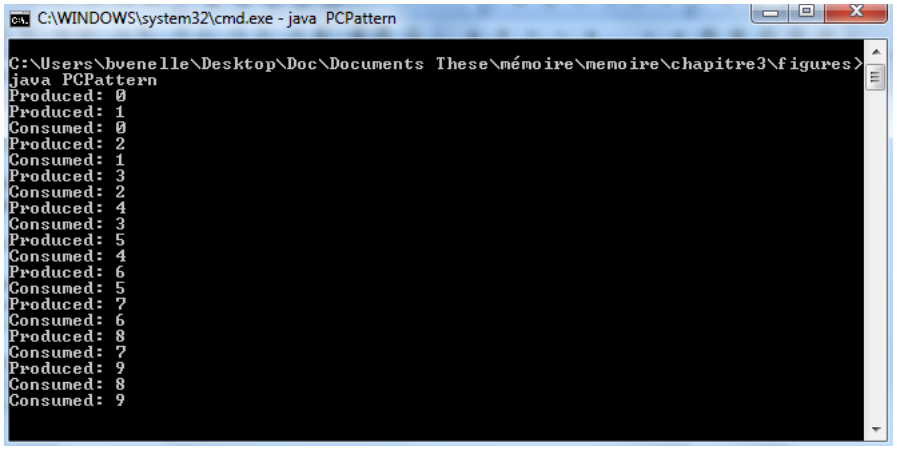



FIGURE 3.14 – Sortie du producteur/consommateur en Java

À l'exécution de ce programme, nous pouvons observer la création de deux fils d'exécutions correspondants respectivement à une activité producteur et une activité consommateur se partageant un même objet de type *Queue* (cf. lignes 12/13). Au moment de l'exécution de ces deux activités, nous pourrons observer les appels à l'objet partagé *sharedQueue* (cf. lignes 33 et 51). Ainsi, les flux d'activité montrent que l'objet partagé *sharedQueue* obtient temporairement le contrôle du système :

$$O \equiv \{o_1, o_2, o_3, o_4\} \quad S \equiv \{o_2, o_3\}$$

$$\begin{aligned}
& m_2^2 \xrightarrow{invoke} m_1^4 \\
\equiv & o_2 \xrightarrow{m_2^2\ invoke\ m_1^4} o_4 \\
\equiv & Contexte(o_2) \xrightarrow{Signature(m_2^2)\ invoke\ Signature(m_1^4)} Contexte(o_4) \\
\equiv & \text{"}prodThread\text{"} \xrightarrow{\text{"}run\text{"}\ invoke\ \text{"}put\text{"}} \text{"}sharedQueue\text{"} \\
\equiv & \text{"}prodThread\text{"} \triangleright\triangleright \text{"}sharedQueue\text{"}
\end{aligned}$$

$$\begin{aligned}
& m_1^4 \xrightarrow{return} m_2^2 \\
\equiv & o_4 \xrightarrow{m_1^4\ return\ m_2^2} o_2
\end{aligned}$$

```java
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.logging.*;

public class PCPattern {

  public static void main(String args[]){
    // Creating shared object
    BlockingQueue sharedQueue = new LinkedBlockingQueue();

    // Creating Producer and Consumer Thread
    Thread prodThread = new Thread(new Producer(sharedQueue));
    Thread consThread = new Thread(new Consumer(sharedQueue));

    // Starting producer and Consumer thread
    prodThread.start();
    consThread.start();
  }
}

class Producer implements Runnable {  // Producer Class in java

  private final BlockingQueue sharedQueue;

  public Producer(BlockingQueue sharedQueue) {
    this.sharedQueue = sharedQueue;
  }

  @Override
  public void run() {
    for(int i=0; i<10; i++) try {
      System.out.println("Produced: " + i);
      sharedQueue.put(i);
    } catch (InterruptedException e) {
      Logger.getLogger("Producer").log(Level.SEVERE, null, e);
    }
  }
}

class Consumer implements Runnable {  // Consumer Class in Java

  private final BlockingQueue sharedQueue;

  public Consumer (BlockingQueue sharedQueue) {
    this.sharedQueue = sharedQueue;
  }

  @Override
  public void run() {
    while(true) try {
      System.out.println("Consumed: " + sharedQueue.take());
    } catch (InterruptedException e) {
      Logger.getLogger("Consumer").log(Level.SEVERE, null, e);
    }
  }
}
```

FIGURE 3.15 – Modèle producteur/consommateur en Java

Cette implémentation du modèle producteur/consommateur [a] s'appuie sur trois objets. Le premier produit des données sous forme de nombre. Un deuxième objet récupère ces données afin de les traiter. Un objet intermédiaire partagé permet l'échange de données entre les deux premiers.

*a.* `http://javarevisited.blogspot.fr/2012/02/producer-consumer-design-pattern-with.html`

$$\equiv Contexte(o_4) \xrightarrow{Signature(m_1^4)\ return\ Signature(m_2^2)} Contexte(o_2)$$
$$\equiv \text{"}sharedQueue\text{"} \xrightarrow{\text{"}put\text{"}\ return\ \text{"}run\text{"}} \text{"}prodThread\text{"}$$
$$\equiv \text{"}sharedQueue\text{"} \triangleright\triangleright \text{"}prodThread\text{"}$$

$$m_2^3 \xrightarrow{invoke} m_2^4$$
$$\equiv o_3 \xrightarrow{m_2^3\ invoke\ m_2^4} o_4$$
$$\equiv Contexte(o_3) \xrightarrow{Signature(m_2^3)\ invoke\ Signature(m_2^4)} Contexte(o_4)$$
$$\equiv \text{"}consThread\text{"} \xrightarrow{\text{"}run\text{"}\ invoke\ \text{"}take\text{"}} \text{"}sharedQueue\text{"}$$
$$\equiv \text{"}consThread\text{"} \triangleright\triangleright \text{"}sharedQueue\text{"}$$

$$m_2^4 \xrightarrow{return} m_2^3$$
$$\equiv o_4 \xrightarrow{m_2^4\ return\ m_2^3} o_3$$
$$\equiv Contexte(o_4) \xrightarrow{Signature(m_2^4)\ return\ Signature(m_2^3)} Contexte(o_3)$$
$$\equiv \text{"}sharedQueue\text{"} \xrightarrow{\text{"}take\text{"}\ return\ \text{"}run\text{"}} \text{"}consThread\text{"}$$
$$\equiv \text{"}sharedQueue\text{"} \triangleright\triangleright \text{"}consThread\text{"}$$

Par ailleurs, sachant que les méthodes *put* et *take* acceptent respectivement un paramètre en entrée et un paramètre en sortie, nous pouvons observer un flux d'information correspondant au fonctionnement du modèle producteur/consommateur. En effet :

$$O \equiv \{o_1, o_2, o3, o_4\} \quad S \equiv \{o_2, o_3\}$$

$$m_2^2 \xrightarrow{invoke} m_1^4$$
$$\equiv o_2 \xrightarrow{m_2^2\ invoke\ m_1^4} o_4$$
$$\equiv Contexte(o_2) \xrightarrow{Signature(m_2^2)\ invoke\ Signature(m_1^4)} Contexte(o_4)$$
$$\equiv \text{"}prodThread\text{"} \xrightarrow{\text{"}run\text{"}\ invoke\ \text{"}put\text{"}} \text{"}sharedQueue\text{"}$$
$$\equiv \text{"}prodThread\text{"} \Longrightarrow \text{"}sharedQueue\text{"}$$

$$m_2^4 \xrightarrow{return} m_2^3$$
$$\equiv o_4 \xrightarrow{m_2^4\ return\ m_2^3} o_3$$
$$\equiv Contexte(o_4) \xrightarrow{Signature(m_2^4)\ return\ Signature(m_2^3)} Contexte(o_3)$$
$$\equiv \text{"}sharedQueue\text{"} \xrightarrow{\text{"}take\text{"}\ return\ \text{"}run\text{"}} \text{"}consThread\text{"}$$
$$\equiv \text{"}sharedQueue\text{"} \Longrightarrow \text{"}consThread\text{"}$$

Dit autrement, nous observons alors un flux d'information de l'objet *prodThread* vers l'objet *consThread* via l'objet *sharedQueue* :

$$"prodThread" \Longrightarrow "sharedQueue" \Longrightarrow "consThread"$$

Plus particulièrement, si l'on s'intéresse à la valeur des données transférées, nous observons que lorsque l'objet *prodThread* produit un entier, ce même entier est automatiquement envoyé à l'objet *consThread* ce qui est conforme au modèle producteur/consommateur :

$$\begin{aligned} "prodThread" &\overset{1}{\Rightarrow} "sharedQueue" \overset{1}{\Rightarrow} "consThread" \\ "prodThread" &\overset{2}{\Rightarrow} "sharedQueue" \overset{2}{\Rightarrow} "consThread" \\ "prodThread" &\overset{3}{\Rightarrow} "sharedQueue" \overset{3}{\Rightarrow} "consThread" \\ &\vdots \end{aligned}$$

Au final, on voit qu'en précisant la nature des flux de chaque interaction nous faisons ressortir des détails de plus haut niveau et que l'on peut contrôler. Nous pourrions ainsi définir des règles qui, par exemple, forcent les données émises par un objet *prodThread* à transiter par un objet *sharedQueue* avant d'être utilisées par un objet *consThread*. Une telle règle offre une bien meilleure précision que de simplement interdire à un objet *prodThread* d'accéder directement à un objet *consThread*. Nous verrons dans la sous-section suivante comment mettre en œuvre une telle logique.

### 3.3.4 Logique de relation

Dans la sous-section précédente, nous avons pris l'exemple du modèle producteur/consommateur qui s'appuie sur trois objets pour transmettre de l'information. Nous y avons alors mis en lumière l'existence d'un flux d'information entre un objet dit *producteur* et un objet dit *consommateur*. En réalité, lorsque l'on s'intéresse à la chronologie des flux, comme nous l'avions implicitement fait dans cet exemple, nous pouvons construire une logique. Cette logique peut utiliser non seulement des relations de flux mais aussi les relations de référence et d'interaction. Ainsi nous pourrons utiliser une logique de flux d'information comme dans [78, 79] mais notre logique va au delà puisqu'elle inclut également des relations de flux d'activité et de données ainsi que les relations de référence et d'interaction.

Afin d'aller plus loin et d'élargir [78, 79], nous proposons de structurer les relations à satisfaire en relations élémentaires (référence, interaction, flux) et de

les décrire par des automates où les états sont les objets à surveiller et les transitions représentent les privilèges requis pour passer d'un objet à l'autre. Nous verrons dans le chapitre suivant comment construire de tels automates.

**Hypothèse 3.3 (Automates de contrôle)** *À partir des travaux de Von Neumann sur les machines universelles, s'il existe un algorithme pour décrire la logique d'une relation inter-objets alors il existe un automate qui implémente cette logique. Si un tel automate existe alors nous serons en mesure de reconnaître et contrôler des relations complexes.*

Il s'agit bien là d'un élargissement des travaux antérieurs sur au moins deux points. D'une part, nous couvrons des contrôles fins entre objets d'un même espace d'adressage. D'autre part, nous proposons une formalisation plus générale et plus simple des relations comme la référence, l'interaction ainsi que les différents types de flux et d'automates. Nous verrons que ces notions couvrent entre autres le cas de Java et le modèle de sécurité de celui-ci tout en permettant une meilleure garantie de sécurité.

## 3.4 Relations spécifiques entre objets et politiques spécialisées

Le modèle de relations que nous avons établi est suffisant pour traiter le cas général. Mais comme nous l'avons vu dans la section 3.2, beaucoup de systèmes à objets apportent leurs lots de spécificités. Cette section propose d'étudier l'impact de ces spécificités sur les relations entre objets afin de formaliser plus précisément ces cas particuliers si nécessaire.

### 3.4.1 Héritage

L'un des concepts clefs des systèmes basés sur les classes est la notion d'héritage. Celle-ci est à la base du mécanisme de polymorphisme qui permet la surcharge et la redéfinition de membres de classes. Il s'agit d'une relation par laquelle une classe réutilise la définition d'une ou plusieurs classes existantes appelées "classes parentes". Cette nouvelle classe représente alors un sous-type de ses classes parentes et par conséquent elle possède les noms de ces dernières. Certains langages comme C++ utilisent le terme de "dérivation" afin de souligner le fait qu'une classe implémente une définition étendue/dérivée des membres des classes parentes.

**Notation 3.24 (Héritage)** *Soit $O$ un système à objets et $c_k$ une classe de $C$.*

1. *On définit l'application Parents qui renvoie l'ensemble des classes parentes d'une classe $c_k$ telle que :*
   $\forall c_k \in C$,

$$\begin{aligned} Parents\colon C &\longrightarrow C \\ c_k &\longmapsto \ \{c_p \in C,\ Noms(c_p) \subset Noms(c_k)\} \end{aligned}$$

2. *On définit la fonction Super qui renvoie une référence sur une classe parente d'une classe $c_k$ telle que :*
   $\forall c_k \in C, \exists c_p \in Parents(c_k)$,

$$\begin{aligned} Super\colon C \times C &\longrightarrow R \\ c_k, c_p &\longmapsto \ ref^{k,p} \in Références(c_k) \end{aligned}$$

3. *On note $c_k \twoheadrightarrow c_p$ l'héritage d'une classe $c_p$ par une classe $c_k$ tel que :*
   $\forall c_k, c_p \in C, c_k \twoheadrightarrow c_p \equiv c_p \in Parents(c_k)$

En pratique, l'héritage est surtout une relation structurelle qui symbolise la spécialisation d'une classe par rapport à une autre et, de fait, elle ne peut être réciproque. Par exemple le type *chat* est une spécialisation du type *animal de compagnie* ce qui se traduit par un lien d'héritage entre ces deux types. L'héritage a un impact sur les politiques de sécurité car un *chat* est aussi un *animal de compagnie* et de fait les privilèges qui s'appliquent pour la classe *animal de compagnie* peuvent aussi s'appliquer à la classe *chat*, si besoin.

Le point positif de cette considération est que l'héritage nous permet alors de calculer les droits d'accès pour tout types d'objet qui ne figureraient pas dans la politique de sécurité. Par exemple, nous pouvons autoriser les objets de type *humain* à appeler la méthode *adopter* des objets de type *animal de compagnie* ; ce qui inclut les objets de type *chat*, *chien*, *furet*, etc. À l'inverse, cette transmission automatique de privilèges via les relations d'héritage peut ne pas être souhaitable. C'est pourquoi, les politiques de sécurité doivent obligatoirement fournir un moyen de désactiver un privilège particulier pour des objets d'un type donné, bloquant ainsi l'héritage du privilège correspondant. D'où la nécessité de supporter les règles de type *deny* de façon à ce que la règle la plus restrictive s'applique [19].

### 3.4.2 Instanciation

L'instanciation et la destruction d'objet sont deux processus nécessaires au cycle de vie des objets d'un système à classes. Le premier fait appel à deux opérations, l'allocation et l'initialisation, qui sont implémentées sous la forme de méthodes particulières ; la destruction de l'objet étant elle aussi implémentée sous la forme d'une méthode de cette classe (cf. figure 3.16).

(a) Construction (b) Initialisation (c) Destruction

FIGURE 3.16 – Cycle de vie d'un objet de type "maison"

L'objet "maison" est d'abord construit afin d'exister dans le système (a). Immédiatement après, celui-ci est initialisé pour être prêt à être utilisé (b). Enfin, cet objet est détruit lorsqu'il devient inutile (c). Chacune de ces étapes est implémentée par trois méthodes particulières : le constructeur, l'initialisateur et le destructeur.

L'objectif d'un constructeur est donc de construire un objet à partir de sa classe. Ainsi lorsque le constructeur est appelé, un nouvel objet est créé dans le système dont la stratégie de construction dépend du système considéré. Dans un système informatique, par exemple, la construction d'un objet se traduit par son allocation en mémoire. On notera que l'existence d'un constructeur peut être implicite, comme c'est le cas avec le langage Java ou explicite comme avec C++ (redéfinition de l'opérateur *new*[7]) ou Python (méthode *___new___*[8]).

L'initialisateur est la toute première méthode à être exécutée par l'objet nouvellement créé. Son but est de fixer les valeurs initiales de l'objet en fonction des différents paramètres passés en entrée. Les relations d'héritage de la classe font qu'un initialisateur donné ne peut initialiser les données des membres hérités. De fait, lorsque un initialisateur est appelé, celui-ci appelle d'abord les initialisateurs définis dans les classes parentes. La majorité des langages à objets définissent un initialisateur explicite par classe comme Python (méthode *___init___*[9]) mais la plupart l'assimile, par abus, au constructeur de la classe comme Java (Méthodes *<init>* et *<clinit>*[10]).

Enfin, le destructeur a pour effet de détruire un objet lorsque celui-ci n'est plus utile pour le système. En pratique, il s'agit de la toute dernière méthode appelée

7. http://www.cplusplus.com/reference/new/operator\%20new/
8. https://docs.python.org/3/reference/datamodel.html?highlight=__new__#object.__new__
9. https://docs.python.org/3/reference/datamodel.html?highlight=__init__#object.__init__
10. http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html#jvms-2.9

par un objet avant qu'il disparaisse du système. Comme pour le constructeur, chaque système définit lui même sa stratégie de destruction d'objets. Des langages de programmation comme Java ou Javascript s'appuient sur un ramasse miettes par exemple. Mais certains langages offrent la possibilité de maîtriser cette opération comme C++ (redéfinition de l'opérateur *delete* [11]) ou Python dans une certaine mesure (méthode ___*del*___ [12]). Du fait des relations d'héritage, lorsqu'un destructeur est appelé, celui-ci appellera aussi les destructeurs des classes parentes de l'objet.

Au final, le processus d'instanciation utilise simplement des méthodes d'objet particulières. De fait notre modèle basé sur la notion de méthodes s'applique et permet de contrôler le cycle de vie des objets sans notation supplémentaire.

### 3.4.3 Mutation

La notion de mutation d'objet est propre aux langages à prototypes. Le plus souvent elle est silencieuse pour l'objet considéré car celui-ci n'a pas connaissance des mutations qu'il subit. Par définition, il s'agit d'une opération basée sur l'insertion, la suppression et la substitution de données qui a pour effet de modifier la définition d'un objet. En particulier, l'objet qui initie la mutation peut induire la création ou la suppression de membres de même qu'un changement de types. En pratique la mutation s'appuie surtout sur l'introspection et l'intercession pour respectivement observer et modifier les données d'un objet.

En pratique, la mutation n'est qu'une opération de lecture/écriture un peu particulière que nous savons décrire avec notre notation. De fait si nous sommes en mesure d'observer ce type d'interaction alors nous pourrons autoriser un objet à muter ou non. Par contre, en termes de contrôle, aucun objet ne pourra posséder de privilèges fixes car ceux-ci sont susceptibles de changer au gré des différentes mutations. Nous pourrions alors imaginer une logique de contrôle supplémentaire basée sur la nature des mutations qu'un objet subit. Une telle approche permet alors d'exprimer les politiques de sécurité en fonction des mutations possibles et non plus en fonction de la nature des objets.

### 3.4.4 Clonage

Par définition, le clonage a pour effet de dupliquer un objet existant pour créer un nouvel objet. Contrairement à l'instanciation, un clone d'objet n'a pas besoin d'être initialisé car celui-ci est une copie conforme de l'objet d'origine. Le

11. `http://www.cplusplus.com/reference/new/operator\%20delete/`
12. `https://docs.python.org/3/reference/datamodel.html?highlight=__del__#object.__del__`

processus fait intervenir deux opérations : l'allocation et la copie. Comme pour l'instanciation, ces deux opérations sont implémentées dans chaque objet sous la forme de méthodes spécialisées telles que la méthode *Clone* en Java [13]. Là encore il n'est pas nécessaire d'introduire de la notation supplémentaire pour contrôler le clonage d'un objet puisque cela repose sur la notation des méthodes que nous savons formaliser et contrôler.

En pratique, le clonage est une relation d'objets qui lie un clone à son prototype. On ne parle plus d'héritage, puisqu'il ne s'agit pas d'une spécialisation, mais de lignage. C'est à dire que tous les objets d'une même lignée se considèrent comme descendants d'un ancêtre commun, connu et nommé [14]. En termes de politique de sécurité il y a donc ici aussi transmission des privilèges d'un prototype à sa lignée. Nous ferons donc le même constat que pour l'héritage : la politique de sécurité doit permettre de désactiver des privilèges précis pour des objets en particulier.

## 3.5 Conclusion

L'objectif de ce chapitre était de comprendre le fonctionnement des systèmes à objets en modélisant les différentes entités qui les composent. Il nous est apparu que les notations déjà existantes étaient trop limitées pour être exploitables. Nous avons donc fait le choix d'en définir une nouvelle mais plus simple à mettre en œuvre car majoritairement basée sur des notations ensemblistes.

Ainsi, nous sommes partis d'une modélisation générale des systèmes à objets et avons projeté ce modèle sur trois systèmes particuliers que sont les langages à classes, les langages à prototypes et les systèmes répartis. Mais plus globalement, la notation que nous proposons supporte des langages au delà des langages à objets. En effet, parmi les exemples que nous avons donnés, certains sont des systèmes qui décrivent des objets mais dont le langage n'est pas objet. Cela signifie que quelque soit le système considéré, s'il est possible d'assimiler les entités qui le composent à des objets, alors notre notation devrait s'appliquer moyennant, peut être, quelques adaptations pour tenir compte des particularités du système en question. Bien entendu, il ne s'agit que d'une hypothèse de travail mais qui semble tout de même se vérifier en pratique.

Nous étions conscients des limites imposées par le théorème de Rice [30] et de fait nous avons orienté notre notation sur l'idée des relations observables dynamiquement et non statiquement. Notre modèle décrit ainsi les relations observables entre objets ; c'est à dire les références, les interactions et les flux.

13. `http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#clone--`
14. Il s'agit de la définition donnée par le Larousse.

La notion de référence n'implique pas nécessairement une opération d'accès car un objet peut être référencé sans jamais être utilisé. Par contre, celle-ci est un prérequis nécessaire avant toute interaction d'un objet avec un autre. Nous sommes arrivés à la même conclusion que Edsger Dijkstra [5] sur la nature anthropomorphique des objets qui, dans notre modèle, est de penser que les objets décident des actions mais que ce sont leurs membres qui les réalisent. Nous avons ainsi pu établir quatre types possibles d'interactions entre membres et pour chacune d'entre elle la possibilité de réaliser une lecture, une écriture ou une exécution. Mais dans les faits les interactions entre objets sont principalement de l'ordre de l'appel de méthode et de l'accès à un champ.

Par ailleurs, nous avons expliqué que l'échange de données entre objets n'était possible que si ces deux objets interagissent l'un avec l'autre. Nous avons montré que cet échange n'allait pas obligatoirement dans le même sens que l'interaction comme c'est le cas avec les opérations de lecture. De même une interaction n'est pas nécessairement synonyme d'échange de données comme lors d'un appel de méthode sans paramètre. Cela nous a alors permis d'établir trois types de flux : le flux d'activités, le flux d'information et le flux de données lorsque l'on s'intéresse à la valeur de l'information échangée.

Ainsi, nous avons considéré que si l'on est dans la capacité technique d'observer ces différentes relations alors nous pouvons contrôler chacune d'entre elles. Pour cela, nous avons proposé d'exprimer les règles de contrôle à l'aide du même formalisme que notre modèle de relations. Par cette approche, les relations observées sont exprimées en fonction des contextes de sécurité et les politiques en fonction des signatures. La signature étant incluse dans l'expression du contexte de sécurité de l'objet, il devient alors facile de réaliser une comparaison entre les deux et donc d'appliquer une politique de sécurité.

Les expériences que nous avons réalisées ont montré la pertinence et l'efficacité de cette approche sur des environnements Java [80, 81]. Malheureusement, comparer les relations une à une avec une politique de sécurité impose d'écrire autant de règles qu'il y a de relations. Au regard du nombre d'objets différents dans un système et de la combinatoire des relations possibles, il est très difficile d'écrire et de maintenir ces politiques.

Par contre, la littérature et notamment [78] montre qu'il est possible de construire une logique de contrôle élaborée à partir de l'observation du flux dans un système. Cette logique a alors pour effet d'augmenter la pertinence des règles de contrôle. À l'inverse des travaux antérieurs [78, 79], nous ne sommes pas contraints par une logique particulière. En effet, nous ne considérons que les opérations élémentaires associant les objets et pouvant être utilisées dans tout langage, toute logique ou toute grammaire d'automates. Ainsi nous réduisons la formalisation à un noyau minimal pouvant être utilisée par tout système logique existant. Nous montrerons l'efficacité de cette approche dans le cas de la logique de contrôle JAAS associée à Java.

Ainsi nous avons eu l'intuition de réduire la logique à un automate d'états. Pour cela nous sommes partis de l'idée d'exprimer chaque flux élémentaire sous la forme d'une capacité permettant au flux observé de progresser dans cet automate ; chaque état dudit automate étant un objet du système. Le chapitre suivant traite des spécifications fonctionnelles nécessaires à un tel contrôle et le dernier présente une application pratique de celui-ci.

Chapitre 4

# Contrôle d'accès basé sur les automates pour Java

## Sommaire



*Même si vous ne faites rien de mal, vous êtes surveillés et enregistrés.*

Edward Snowden [82]

Le chapitre précédent propose un formalisme pour modéliser les systèmes à objets et plus particulièrement représenter les relations entre objets. Nous avons établi qu'il existait trois relations possibles que sont la référence, l'interaction et la logique de flux. Ainsi la référence est le fait, pour un objet, de désigner et d'obtenir un accès à un autre objet du système dans le but éventuel de l'utiliser. Ainsi, l'interaction est une action effective permettant au premier objet d'utiliser l'objet référencé. Enfin, nous avons expliqué qu'il était possible de construire une logique portant sur les échanges d'informations entre objets et nous avons émis l'hypothèse de mettre en œuvre des automates pour implémenter cette logique. Nous commencerons donc ce chapitre par étudier cette hypothèse.

Le théorème de Rice [30] montre qu'il n'est pas possible de déterminer à l'avance les relations entre objets. C'est pour cette raison que notre approche est centrée sur les relations observables d'un système à objets en cours de fonctionnement. Or, nous savons que les attaques sur Java que nous avons identifiées (cf. section 2.2) peuvent avoir un impact sur les objets de la JVM et plus particulièrement sur les relations entre ces objets. C'est pourquoi, nous souhaitons confronter le modèle que nous avons défini avec le modèle d'attaque établi dans l'état de l'art.

Par ailleurs, il faut proposer un moyen de garantir la politique de sécurité qui utilise notre modèle grâce à un modèle de contrôle obligatoire (MAC) à grain fin. Nous sommes conscients que la définition de politiques à grain fin est très difficile et limite l'utilisation réaliste de ce type d'approche. Néanmoins des travaux récents [3,67] laissent entendre qu'il ne s'agit pas d'un problème insoluble. C'est pourquoi nous proposons dans ce chapitre de projeter les fondamentaux du contrôle d'accès obligatoire à grain fin sur les systèmes à objets et d'en proposer des spécifications fonctionnelles pour Java. Nous terminerons ce chapitre sur une méthode pour transcrire les politiques JAAS en politique obligatoire à grain fin.

La figure 4.1 représente les trois principales contributions de ce chapitre.

## 4.1 JAAS Vs Inspection de pile par automates

Le cas d'étude proposé par [83] est idéal selon nous pour traiter les problèmes de sécurité dans une machine virtuelle Java. Notamment, il met en lumière la possibilité pour un objet non-privilégié de réaliser un accès privilégié sur un objet protégé car le scénario proposé se place dans le cas où, pour des raisons fonctionnelles, une élévation de privilèges est nécessaire. Néanmoins, celui-ci ayant un but éducatif, il ne reflète pas entièrement la réalité. C'est pourquoi nous commencerons cette section par présenter un cas d'étude légèrement enrichi par rapport à l'original.

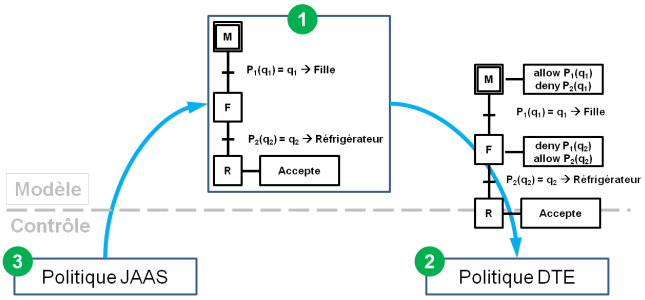



FIGURE 4.1 – Approche globale et principales contributions de ce chapitre

① Nous utilisons des automates pour décrire la dynamique des relations et ainsi exprimer des objectifs de sécurité avancés. ② Nous garantissons ensuite ces besoins de façon obligatoire via une projection dynamique de ces automates en une politique de sécurité centrée sur les seuls objets pertinents à contrôler. Cela a notamment pour effet de réduire mécaniquement le "bruit" de la politique, ce qui améliore d'autant la qualité de la protection. ③ Enfin, nous savons réutiliser les objectifs de sécurité existants en retrouvant dynamiquement les automates correspondants à partir d'une politique. Ces trois contributions conjuguées permettent ainsi de garantir la politique JAAS de n'importe quelle application Java de façons transparente, obligatoire et totalement autonome.

Grâce à cela, nous pourrons montrer que JAAS peut facilement être mis en défaut lorsque il s'agit de mettre des élévations de privilèges. Nous proposerons ensuite une approche alternative qui est d'implémenter la logique de contrôle JAAS à l'aide d'automates. Pour cela, nous expliquerons comment construire un automate de contrôle dans le cas général grâce au modèle que nous avons défini dans le chapitre précédent. Puis nous montrerons comment utiliser ces automates pour reconnaître des flux par inspection de pile.

### 4.1.1 Présentation du cas d'étude

Nous proposons un cas d'étude largement inspiré de [83] qui repose sur trois objets : un maçon, une petite fille et un réfrigérateur. L'objectif est de contrôler l'accès à l'objet réfrigérateur selon les mêmes termes définis dans le cas d'étude original 4.2. Nous introduisons un quatrième objet qui est l'apprenti du maçon dont le seul privilège est de pouvoir dialoguer avec le maçon. Nous identifierons ces quatre objets par $o_a$, $o_r$, $o_f$ et $o_m$ respectivement.

$$O \equiv \{o_a, o_r, o_f, o_m\}$$

$$\begin{aligned} Types(o_a) &\equiv \text{"}Apprenti\text{"} \quad Types(o_m) \equiv \text{"}Maçon\text{"} \\ Types(o_f) &\equiv \text{"}Fille\text{"} \quad Types(o_r) \equiv \text{"}Réfrigérateur\text{"} \end{aligned}$$

Selon ce cas d'étude nous souhaitons garantir quatre objectifs de sécurité que nous décrivons sous la forme d'une politique de sécurité utilisant notre modèle.

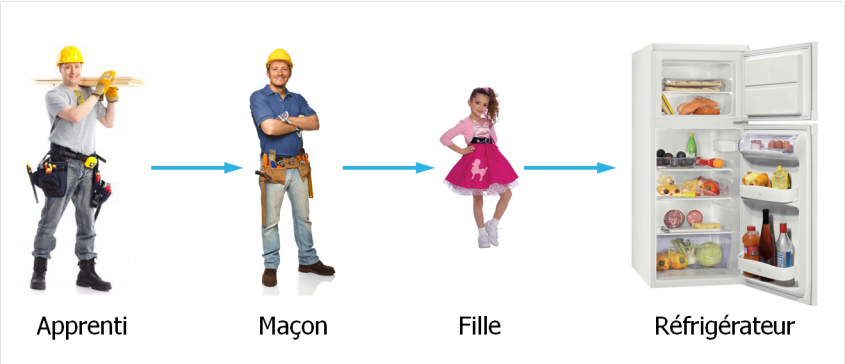



FIGURE 4.2 – Illustration du cas d'étude proposé

Nous souhaitons contrôler l'accès au réfrigérateur. La petite fille est autorisée à y prendre une boisson. Ni le maçon, ni l'apprenti ne peuvent se servir eux mêmes. La petite fille peut donner une boisson au maçon lorsque son travail est terminé.

La règle 4.1 traduit l'idée que le maçon peut demander une boisson à la petite fille qui à son tour en demande une au réfrigérateur. La règle 4.2 correspond à l'obtention de la dite boisson. La règle 4.3 autorise l'apprenti à interagir avec le maçon. Enfin, La règle 4.4 interdit explicitement à l'apprenti d'accéder au réfrigérateur.

$$\begin{aligned} &allow\ Signature(o_m) \Longrightarrow Signature(o_f) \Longrightarrow Signature(o_r) \\ \equiv &allow\ "Maçon" \Longrightarrow "Fille" \Longrightarrow "Réfrigérateur" \end{aligned} \tag{4.1}$$

$$\begin{aligned} &allow\ Signature(o_r) \overset{bière}{\Longrightarrow} Signature(o_f) \overset{bière}{\Longrightarrow} Signature(o_m) \\ \equiv &allow\ "Réfrigérateur" \overset{bière}{\Longrightarrow} "Fille" \overset{bière}{\Longrightarrow} "Maçon" \end{aligned} \tag{4.2}$$

$$\begin{aligned} &allow\ Signature(o_a) \xrightarrow{invoke} Signature(o_m) \\ \equiv &allow\ "Apprenti" \xrightarrow{invoke} "Maçon" \end{aligned} \tag{4.3}$$

$$\begin{aligned} &deny\ Signature(o_a) \longrightarrow Signature(o_r) \\ \equiv &deny\ "Apprenti" \longrightarrow "Réfrigérateur" \end{aligned} \tag{4.4}$$

Ajoutons également deux objets anonymes, $o_x$ et $o_y$, dont le but sera de générer du bruit dans l'observation des relations entre objets du système. Notre but sera de contrôler par inspection de pile la séquence d'interactions suivante qui traduit le fait que l'apprenti demande une bière au maçon, qui à son tour la demande à la petite file qui l'obtient du réfrigérateur.

$$o_a \xrightarrow{invoke} o_m \xrightarrow{invoke} o_x \xrightarrow{invoke} o_f \xrightarrow{invoke} o_y \xrightarrow{invoke} o_r$$

### 4.1.2 Approche JAAS : la difficulté de l'élévation de privilège

Nous savons que JAAS utilise le principe de l'inspection de pile pour vérifier les privilèges des objets présents dans une pile d'exécution afin d'autoriser ou non l'accès à un objet protégé. Nous allons montrer ici, avec notre cas d'étude, que l'implémentation par JAAS de l'inspection de pile n'est pas sans défauts. Pour les besoins de chacun de ces scénarios, nous considérerons la politique JAAS suivante exprimée à l'aide de notre modèle :

$$allow\ Signature(o_m) \xrightarrow{invoke} Signature(o_f) \tag{4.5}$$

$$\equiv allow\ \text{"}Maçon\text{"} \xrightarrow{invoke} \text{"}Fille\text{"} \tag{4.6}$$

$$allow\ Signature(o_f) \xrightarrow{invoke} Signature(o_r) \tag{4.7}$$

$$\equiv allow\ \text{"}Fille\text{"} \xrightarrow{invoke} \text{"}Réfrigérateur\text{"} \tag{4.8}$$

$$allow\ Signature(o_a) \xrightarrow{invoke} Signature(o_m) \tag{4.9}$$

$$\equiv allow\ \text{"}Apprenti\text{"} \xrightarrow{invoke} \text{"}Maçon\text{"} \tag{4.10}$$

**Scénario : Le maçon demande à la fille de prendre une bière dans le frigo**

Ignorons dans un premier temps l'existence de l'objet apprenti et plaçons nous dans la même situation que le troisième scénario de [83] afin d'analyser le fonctionnement "normal" de JAAS. Nous considérerons donc, pour l'instant, juste la séquence :

$$o_m \xrightarrow{invoke} o_x \xrightarrow{invoke} o_f \xrightarrow{invoke} o_y \xrightarrow{invoke} o_r$$

Les objets de types *Réfrigérateur* font un appel explicite à JAAS pour réaliser une inspection de pile et ainsi vérifier que tous les objets appelant ont le droit de prendre une boisson. Ainsi, selon cette séquence d'interactions, JAAS analyserait les privilèges de l'objet $o_y$, puis de l'objet $o_f$, puis de $o_x$, et enfin de $o_m$. Par optimisation, JAAS ignore en réalité les objets n'ayant pas de domaine de protection, c'est à dire dont le contexte de sécurité et/ou les privilèges ne seraient pas connus. Cette méthode permet à JAAS d'éliminer le "bruit" de son processus d'inspection et de fait la séquence d'interaction observée est perçue par JAAS comme :

$$o_m \xrightarrow{invoke} o_f \xrightarrow{invoke} o_r$$

Or, selon la règle 4.6 de la politique JAAS considérée, un objet de type *Maçon* possède le privilège d'accéder à un objet de type *Fille* mais pas de type *Réfrigérateur*. De fait, JAAS interdira l'accès à l'objet $o_r$ et bloquera la dernière interaction de cette séquence. Mais cela va à l'encontre de l'objectif fonctionnel du scénario qui est d'autoriser un objet de type *Maçon* à accéder au *Réfrigérateur* en passant par un objet *Fille*.

**Scénario : Le maçon demande à la fille de prendre une bière dans le frigo de façon privilégiée**

Ignorons encore l'existence de l'objet apprenti mais plaçons nous cette fois dans la même situation que le cinquième scénario de [83] et conservons la séquence d'interactions précédente :

$$o_m \xrightarrow{invoke} o_f \xrightarrow{invoke} o_r$$

Comme expliqué par les auteurs de [83], la documentation Java [1] fournit des éléments de programmation pour marquer l'objet *Fille* comme privilégié, ordonnant ainsi à JAAS une élévation de privilèges dans la séquence d'interactions observée. En réalité, cette technique force JAAS à interrompre son inspection lorsqu'il atteint le niveau de l'objet *Fille*. Les bonnes pratiques imposent alors de vérifier au niveau de l'objet *Fille* que l'élévation de privilèges demandée est légitime ce qui se traduit en pratique par une inspection de pile en deux temps :

$$\overbrace{o_m \xrightarrow{invoke} o_f}^{1^{\text{ère}}\ inspection} \xrightarrow{invoke} o_r \qquad \underbrace{o_f \xrightarrow{invoke} o_r}_{2^{\text{ème}}\ inspection}$$

La première inspection révèle que l'objet $o_m$ possède le privilège nécessaire pour accéder à l'objet $o_f$ selon la règle 4.6. En accord avec la règle 4.8, la seconde inspection valide également l'accès de l'objet $o_f$ à l'objet $o_r$. Ces deux autorisations combinées permettent alors au maçon de demander une boisson à la fille qui à son tour peut la demander au réfrigérateur sans pour autant autoriser un accès direct du maçon au réfrigérateur.

Intuitivement nous pourrions penser que la règle 4.1 est vérifiée. Cependant JAAS vérifie dans le cas présent la règle *allow* "*Maçon*" $\xrightarrow{invoke}$ "*Fille*" puis la règle *allow* "*Fille*" $\xrightarrow{invoke}$ "*Réfrigérateur*" indépendamment, ce qui ne garantit pas la règle 4.1.

Mais le plus important à noter est que l'élévation de privilèges au niveau de l'objet *Fille* a désactivé la protection qu'offre JAAS sur une partie de la pile d'exécution. Si à première vue cela ne semble pas poser de problèmes, nous verrons avec notre objet apprenti que ce n'est pas le cas.

1. `http://docs.oracle.com/javase/7/docs/technotes/guides/security/doprivileged.html`

**Scénario : L'apprenti demande une bière au maçon**

Réintroduisons donc l'objet apprenti dans notre séquence d'interactions telle que :

$$o_a \xrightarrow{invoke} o_m \xrightarrow{invoke} o_f \xrightarrow{invoke} o_r$$

L'objet fille étant toujours privilégié, l'inspection de pile par JAAS se ferait alors en trois temps ; d'abord la règle 4.3 puis la règle 4.1 :

$$\overbrace{\underbrace{o_a \xrightarrow{invoke} o_m}_{1^{\text{ère}}\ inspection} \xrightarrow{invoke} \underbrace{o_f \xrightarrow{invoke} o_r}_{3^{\text{ème}}\ inspection}}^{2^{\text{ème}}\ inspection}$$

En accord avec la règle 4.10, la première inspection de pile révèle que l'objet *Apprenti* peut effectivement accéder à l'objet *Maçon*. Par contre la seconde inspection de pile montre que l'objet *Apprenti* ne possède pas le privilège pour accéder à l'objet *Fille* et, de fait, JAAS bloquera cet appel. La troisième inspection de pile n'a pas lieu puisque le programme ne s'exécute pas jusque là. De fait, ce comportement est conforme avec ce que l'on attend de JAAS.

**Scénario : L'apprenti demande une bière au maçon de façon privilégiée**

Conservons notre objet apprenti ainsi que notre séquence d'interactions :

$$o_a \xrightarrow{invoke} o_m \xrightarrow{invoke} o_f \xrightarrow{invoke} o_r$$

Supposons que l'objet *Maçon* soit privilégié ; c'est à dire qu'il implémente un bloc *doPrivileged* comme spécifié dans la documentation Java [2]. Puisque cela a pour effet de désactiver l'inspection de pile au niveau de l'objet *Maçon*, Nous obtenons une inspection de pile toujours en trois temps mais bien différente :

$$\underbrace{o_a \xrightarrow{invoke} \overbrace{o_m \xrightarrow{invoke} o_f}^{2^{\text{ème}}\ inspection}}_{1^{\text{ère}}\ inspection} \xrightarrow{invoke} \underbrace{o_r}_{3^{\text{ème}}\ inspection}$$

2. `http://docs.oracle.com/javase/7/docs/technotes/guides/security/doprivileged.html`

Dans ce cas précis, la première inspection autorise l'objet *Apprenti* à accéder à l'objet *Maçon* conformément à la règle 4.10. La seconde inspection s'arrête au niveau de l'objet *Maçon* du fait de l'élévation de privilèges et, conformément aux règles 4.6 et 4.8, JAAS autorise donc cet objet à accéder au réfrigérateur par l'entremise de l'objet *Fille*. Or indirectement, c'est comme si l'apprenti avait demandé une bière au maçon, qui lui-même l'avait demandé à la fille qui l'a obtenu du réfrigérateur alors même que le besoin de sécurité représenté par la règle 4.4 l'interdit ! Résultat : la politique de sécurité est contournée et les besoins ne sont pas satisfaits.

Lorsque l'on analyse les raisons de cette mise en défaut de JAAS, on doit d'abord remarquer que la règle 4.4 ne peut être exprimée avec le langage de politique JAAS car celui-ci ne supporte pas les règles de type *deny*. Pour rappel, une règle de ce type interdit explicitement les accès correspondant même si une règle de type *allow* l'y autorise. Ce qui est différent d'un *deny* implicite lors de l'absence de règles *allow* comme réalisée par JAAS.

Plus généralement, la séquence d'interactions que nous avons considérée avec l'apprenti, le maçon, la fille et le réfrigérateur correspond à une élévation de privilèges via plusieurs objets privilégiés intermédiaires (ici le maçon et la fille). Or ce type d'élévation de privilèges est difficilement maîtrisable lorsque l'on suit une approche purement par programmation comme le fait JAAS. Pour bien faire, il aurait fallu que l'élévation de privilèges se décide par une politique de sécurité plus globale que simplement localisée sur un objet en particulier. Intuitivement on se rend bien compte qu'il est très difficile pour l'objet fille de déterminer si la demande provient effectivement du maçon ou bien d'un autre objet qui utiliserait le maçon comme intermédiaire car, de sa vision, la demande provient toujours du maçon.

Pour information, cette construction de pile d'exécution qui met en échec JAAS est caractéristique des principaux exploits [3] Java où l'apprenti est une applet, le maçon une classe vulnérable, la petite fille une classe système. Nous invitons nos lecteurs à regarder le programme Java que nous avons mis en annexe B et qui implémente ce cas d'étude.

### 4.1.3 Construction d'un automate de contrôle

Dans le chapitre précédent, nous avions émis l'hypothèse d'utiliser des automates pour implémenter une logique de contrôle capable de reconnaître des relations complexes pour mieux les contrôler. Le cas d'étude que nous proposons ici s'inscrit dans ce cas de figure où l'objectif est donc d'apporter *a minima* le même niveau de sécurité que JAAS mais surtout d'aller au delà de ce que

3. Il s'agit d'un élément de programme dont le rôle est d'exploiter une vulnérabilité.

JAAS est capable de garantir. Nous reprenons donc ici les règles considérées précédemment :

$$allow\ "Maçon" \Longrightarrow "Fille" \Longrightarrow "Réfrigérateur" \quad (4.1)$$
$$allow\ "Réfrigérateur" \overset{bière}{\Longrightarrow} "Fille" \overset{bière}{\Longrightarrow} "Maçon" \quad (4.2)$$
$$allow\ "Apprenti" \xrightarrow{invoke} "Maçon" \quad (4.3)$$
$$deny\ "Apprenti" \longrightarrow "Réfrigérateur" \quad (4.4)$$

Nous avions remarqué dans le chapitre précédent qu'une règle exprimée dans notre langage représente la progression d'une relation, tel qu'un flux, au travers de plusieurs objets parfaitement identifiés. Par exemple, la règle 4.1 autorise la progression d'un flux d'information depuis un objet de type *Maçon* vers un objet de type *Réfrigérateur* au travers d'un objet de type *Fille*. Notre idée est donc de décomposer chacune de ces règles en un ensemble d'étapes et de conditions à vérifier pour que la relation progresse d'objet en objet.

Or chacune des règles à transcrire met en lumière les objets importants à surveiller et donc les différentes étapes de l'automate correspondant. Sachant que ces règles de contrôle sont exprimées en fonction de signatures d'objet, les étapes seront également fonction de ces signatures. De même, chaque relation exprimée traduit une condition de transition, c'est à dire le privilège qu'un objet à une étape donnée doit posséder pour que le flux progresse vers l'étape suivante. L'automate pouvant alors décrire soit une autorisation soit une interdiction.

En suivant l'algorithme 1 nous pouvons alors calculer tous les éléments dont nous avons besoin pour construire nos automates. Il est d'ailleurs facile de vérifier qu'en calculant $P_2 \circ P_1("Maçon")$, par exemple, nous retrouvons bien le flux d'information exprimé dans la règle 4.1. Ainsi, Décomposons les règles 4.1 à 4.4 en relations élémentaires applicables à un *Sujet* $q$ quelconque :

$$P_1(q) = q \Longrightarrow "Fille"$$
$$P_2(q) = q \Longrightarrow "Réfrigérateur"$$
$$P_3(q) = q \overset{bière}{\Longrightarrow} "Fille"$$
$$P_4(q) = q \overset{bière}{\Longrightarrow} "Maçon"$$
$$P_5(q) = q \xrightarrow{invoke} "Maçon"$$
$$P_6(q) = q \longrightarrow "Réfrigérateur"$$

Grâce à cela, nous pouvons alors décrire un automate par un ensemble $Q$ d'états

**Entrées** : Un ensemble de besoins de sécurité exprimés par des relations inter-objets.
**Données** : Soit l'ensemble $Q$ des étapes connues et $q_k \equiv Signature(o_k)$ un élément de cet ensemble;
Soit l'ensemble $P$ des privilèges connus et $P_k(q)$ un élément de cet ensemble;
**Résultat** : Un automate par règle avec étapes et transitions. Un ensemble de privilèges.

**pour chaque** *règle* **faire**
  **pour chaque** *signature d'objet* **faire**
    Créer une étape $q_i$ ($q_0$ étant l'étape initiale);
    Ajouter $q_i$ à l'ensemble $Q$;
  **fin**

  **pour chaque** *relation élémentaire* **faire**
    **si** *relation élémentaire* $\equiv q_i \dashrightarrow q_j$ **alors**
      Ajouter à $P$ le privilège $P_k(q) \equiv q \dashrightarrow q_j$;
    **sinon si** *relation élémentaire* $\equiv q_i \longrightarrow q_j$ **alors**
      Ajouter à $P$ le privilège $P_k(q) \equiv q \longrightarrow q_j$;
    **sinon si** *relation élémentaire* $\equiv q_i \triangleright\triangleright q_j$ **alors**
      Ajouter à $P$ le privilège $P_k(q) \equiv q \triangleright\triangleright q_j$;
    **sinon si** *relation élémentaire* $\equiv q_i \Longrightarrow q_j$ **alors**
      Ajouter à $P$ le privilège $P_k(q) \equiv q \Longrightarrow q_j$;
    **sinon si** *relation élémentaire* $\equiv q_i \overset{data}{\Longrightarrow} q_j$ **alors**
      Ajouter à $P$ le privilège $P_k(q) \equiv q \overset{data}{\Longrightarrow} q_j$;
    **sinon**
      Ajouter à $P$ le privilège $P_k(q)$ correspondant à la relation exprimée;
    **fin**

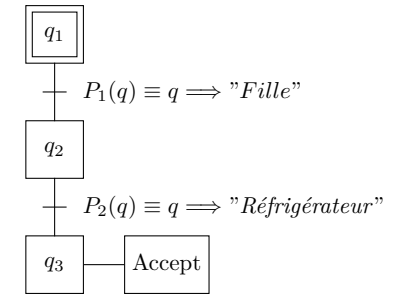
    Créer une transition entre les étapes $q_i$ et $q_j$;
    Ajouter la condition $P_k(q)$ à cette transition;
  **fin**

  **si** *la règle est de type "allow"* **alors**
    Créer une action finale qui accepte la relation reconnue;
  **sinon si** *la règle est de type "deny"* **alors**
    Créer une action finale qui rejette la relation reconnue;
  **sinon**
    Créer une action finale correspondant à l'objectif de la règle;
  **fin**
**fin**

**Algorithme 1 :** Décomposition d'une règle de contrôle en un automate

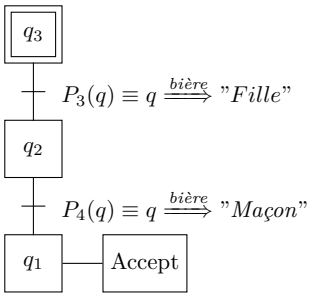
et un ensemble $P$ de transitions, soit la figure 4.3 :

$$
\begin{aligned}
Q \equiv & \{ \\
& q_1 = \text{"}Maçon\text{"}, q_2 = \text{"}Fille\text{"}, \\
& q_3 = \text{"}Réfrigérateur\text{"}, q_4 = \text{"}Apprenti\text{"} \\
& \} \\
P \equiv & \{ \\
& P_1(q_1) = q_1 \Longrightarrow q_2 = \text{"}Maçon\text{"} \Longrightarrow \text{"}Fille\text{"}, \\
& P_2(q_2) = q_2 \Longrightarrow q_3 = \text{"}Fille\text{"} \Longrightarrow \text{"}Réfrigérateur\text{"}, \\
& P_3(q_3) = q_3 \overset{bière}{\Longrightarrow} q_2 = \text{"}Réfrigérateur\text{"} \overset{bière}{\Longrightarrow} \text{"}Fille\text{"}, \\
& P_4(q_2) = q_2 \overset{bière}{\Longrightarrow} q_1 = \Longrightarrow \text{"}Fille\text{"} \overset{bière}{\Longrightarrow} \text{"}Maçon\text{"}, \\
& P_5(q_4) = q_4 \xrightarrow{invoke} q_1 = \text{"}Apprenti\text{"} \xrightarrow{invoke} \text{"}Maçon\text{"}, \\
& P_6(q_4) = q_4 \longrightarrow q_3 = \text{"}Apprenti\text{"} \longrightarrow \text{"}Réfrigérateur\text{"} \\
& \}
\end{aligned}
$$



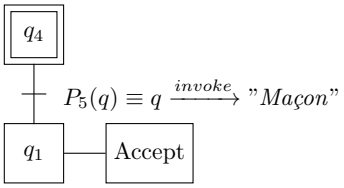

*allow* "Maçon" $\Longrightarrow$ "Fille" $\Longrightarrow$ "Réfrigérateur"

(a) Automate 4.1



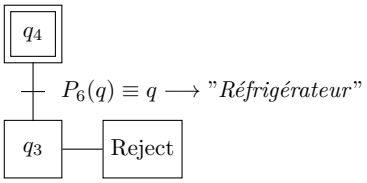

*allow* "Réfrigérateur" $\overset{bière}{\Longrightarrow}$ "Fille" $\overset{bière}{\Longrightarrow}$ "Maçon"

(b) Automate 4.2




*allow* "Apprenti" $\xrightarrow{invoke}$ "Maçon"

(c) Automate 4.3




*deny* "Apprenti" $\longrightarrow$ "Réfrigérateur"

(d) Automate 4.4

FIGURE 4.3 – Automates Grafcet du cas d'étude

Bien entendu, il est nécessaire de donner à chaque objet les permissions requises pour que les flux progressent dans leurs automates respectifs. Par exemple, dans l'automate 4.2, pour que le flux de données progresse de l'étape $q_3$ à $q_2$ il faut que la condition $P_3(q)$ soit remplie. C'est à dire que tout objet correspondant

à l'état $q_3$ doit être en mesure d'accéder aux objets correspondant à l'état $q_2$ et uniquement au travers de la relation $P_3(q)$ exprimée dans la condition. Plus simplement, si nous donnons le privilège $P_3(q)$ aux objets de l'étape $q_3$, nous permettrons les flux de données de la forme $P_3(q_3) \equiv$ "*Réfrigérateur*" $\stackrel{bière}{\Longrightarrow}$ "*Fille*" de progresser vers l'étape $q_2$. Nous pouvons tenir le même raisonnement pour toutes les étapes non finales de chacun de ces automates.

**Entrées** : Un automate et l'ensemble de tous les privilèges connus.
**Résultat** : Un automate avec une politique locale à chaque étape.

**pour chaque** *étape* $q_i$ *non finale* **faire**
  **pour chaque** *privilège* $P_k(q)$ *connu* **faire**
    **si** $P_k(q_i)$ *conditionne une transition sortante* **alors**
      Ajouter une action "*explicit allow* $P_k(q_i)$" à l'étape $q_i$;
    **sinon**
      Ajouter une action "*implicit deny* $P_k(q_i)$" à l'étape $q_i$;
    **fin**
  **fin**
**fin**

**Algorithme 2 :** Définition de la politique locale à chaque étape non finale d'un automate de contrôle

En pratique cela se traduit par créer des actions aux étapes non finales des automates dont l'objectif est de donner des privilèges précis via une règle explicite de type *allow*. De plus si l'on applique le principe de *tout ce qui n'est pas explicitement autorisé est implicitement interdit*, nous devons dès lors créer également des actions similaires pour désactiver les privilèges non nécessaires. C'est à dire que tout privilège de l'ensemble $P$ qui ne représente pas une condition de transition devient automatiquement une règle implicite de type *deny*. En suivant l'algorithme 2 nous pouvons alors définir les politiques locales à chaque étape non finale de ces automates (cf. figure 4.4).

Par contre, une telle construction peut engendrer des conflits de règles entre celles de type *allow* et celles de type *deny*. C'est pourquoi il est nécessaire de définir un ordre de priorité afin de connaître quelles règles l'emportent sur quelles autres.

**Suggestion 4.1 (Ordre de priorité des règles)** *Quelque soit la relation considérée,*

1. *En application du principe de moindre privilège, on favorise toujours la règle la plus restrictive.*
2. *En application du principe de tout ce qui n'est pas autorisé est interdit, on favorise toujours la règle la plus explicite.*
3. *En conformité avec les deux principes précédents, nous définissons l'ordre de priorité des règles de contrôle comme suit :*

$$\textit{implicit allow} < \textit{implicit deny} < \textit{explicit allow} < \textit{explicit deny}$$

Au final nous obtenons un ensemble d'automates, un par règle, où les actions décrites dans chaque étape enrichissent la politique de contrôle des objets traversés par la relation exprimée dans la règle. Nous pouvons vérifier l'exactitude de cette transformation en remplaçant chaque étape par la signature de l'objet exprimé par la règle puis de calculer la politique donnée par chaque action. Nous pouvons d'ailleurs remarquer dans les automates 4.1 et 4.2 de la figure 4.4 deux règles, *deny* $P_1(q_2)$ et *deny* $P_3(q_2)$, qui renforcent l'objectif d'un flux uniquement en transit dans les objets de type *Fille*.

Nous verrons en suivant comment utiliser ces d'automates pour améliorer l'inspection de pile.

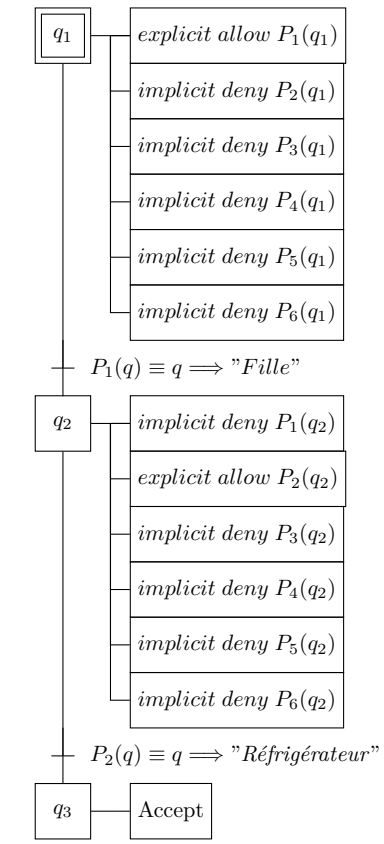
### 4.1.4 Reconnaissance de flux par inspection de pile

Avec notre cas d'étude, nous avons introduit deux objets dont le but est de générer du bruit. De façon générale, ce que nous qualifions de "bruit" désigne tous les éléments du système, que ce soit des objets ou des relations, qui ne sont pas pertinents à surveiller mais pour lesquels il faut tout de même apporter une décision de contrôle d'accès car ceux-ci sont observables et donc contrôlés. Il s'agit là de la faiblesse principale des approches obligatoires puisque, par nature, elles reposent sur l'idée de tout contrôler y compris ce qui n'est pas pertinent et cela impacte automatiquement la difficulté de définir des politiques de sécurité ainsi que leurs tailles. De fait, le bruit perçu lors du contrôle est indirectement reporté sur les politiques de sécurité du système qui deviennent donc soit illisibles car comportant un trop grand nombre de règles non-pertinentes, soit trop permissives pour éviter l'explosion de la politique.

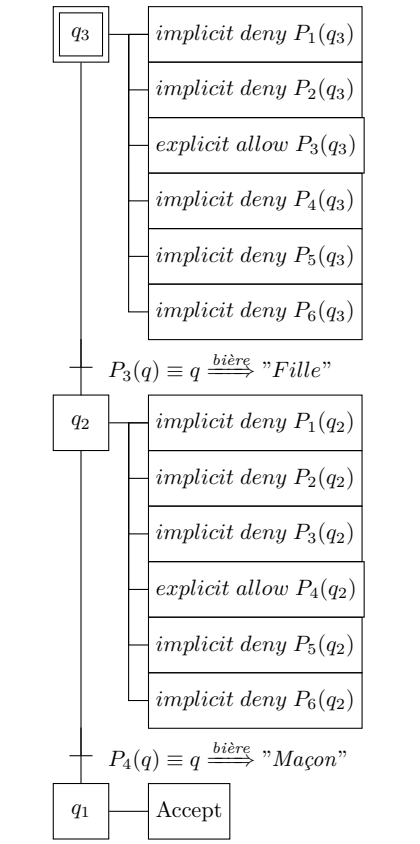
Dans la section précédente, nous avions fait remarquer que si JAAS n'était pas en mesure de calculer le contexte de sécurité d'un objet donné et encore moins ses privilèges, cet objet était automatiquement écarté du processus de contrôle (cf. listing 4.5). Ainsi, dans la séquence d'interactions que nous considérons depuis le début, les objets $o_x$ et $o_y$ sont considérés comme du bruit par JAAS et sont donc écartés du processus de contrôle.

$$\overbrace{o_a \xrightarrow{invoke} \underbrace{o_m}_{2^{\text{ème}}\ analyse}}^{1^{\text{ière}}\ analyse} \xrightarrow{invoke} \overbrace{o_x \xrightarrow{invoke} \underbrace{o_f}_{4^{\text{ème}}\ analyse}}^{3^{\text{ème}}analyse} \xrightarrow{invoke} \overbrace{o_y}^{5^{\text{ème}}analyse} \xrightarrow{invoke} o_r$$

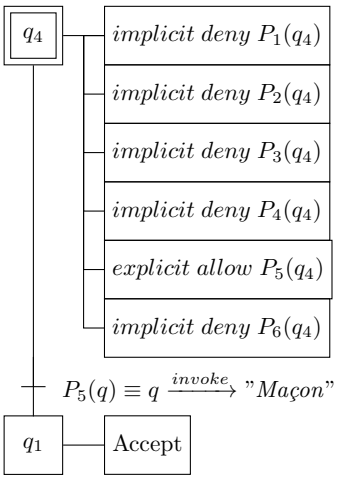
Cependant, nous ne souhaitons pas dans notre cas écarter ces objets car les flux qui les traversent deviendraient alors difficilement qualifiables. Nous pourrions éventuellement suivre la méthode proposée par [78] basée sur une logique dédiée mais introduisant une forte combinatoire. Afin de permettre à l'approche de passer à l'échelle, nous proposons d'ignorer ces objets "parasites" sans pour



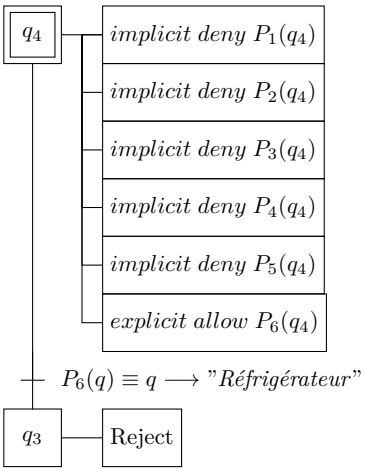

(a) Automate 4.1




(b) Automate 4.2




(c) Automate 4.3




(d) Automate 4.4

FIGURE 4.4 – Automates Grafcet enrichi du cas d'étude

```
1  JVM_ENTRY(jobject, JVM_GetStackAccessControlContext(JNIEnv *env, jclass cls))
2    JVMWrapper("JVM_GetStackAccessControlContext");
3    if (!UsePrivilegedStack) return NULL;
4
5    ResourceMark rm(THREAD);
6    GrowableArray<oop>* local_array = new GrowableArray<oop>(12);
7    JvmtiVMObjectAllocEventCollector oam;
8
9    // count the protection domains on the execution stack. We collapse
10   // duplicate consecutive protection domains into a single one, as
11   // well as stopping when we hit a privileged frame.
12
13   // Use vframeStream to iterate through Java frames
14   vframeStream vfst(thread);
15
16   oop previous_protection_domain = NULL;
17   Handle privileged_context(thread, NULL);
18   bool is_privileged = false;
19   oop protection_domain = NULL;
20
21   for(; !vfst.at_end(); vfst.next()) {
22     // get method of frame
23     methodOop method = vfst.method();
24     intptr_t* frame_id   = vfst.frame_id();
25
26     // check the privileged frames to see if we have a match
27     if (thread->privileged_stack_top() && thread->privileged_stack_top()->
           frame_id() == frame_id) {
28       // this frame is privileged
29       is_privileged = true;
30       privileged_context = Handle(thread, thread->privileged_stack_top()->
             privileged_context());
31       protection_domain  = thread->privileged_stack_top()->protection_domain
             ();
32     } else {
33       protection_domain = instanceKlass::cast(method->method_holder())->
             protection_domain();
34     }
35
36     if ((previous_protection_domain != protection_domain) && (
           protection_domain != NULL)) {
37       local_array->push(protection_domain);
38       previous_protection_domain = protection_domain;
39     }
40
41     if (is_privileged) break;
42   }
43
44   // [...]
45
46   oop result = java_security_AccessControlContext::create(h_context,
         is_privileged, privileged_context, CHECK_NULL);
47
48   return JNIHandles::make_local(env, result);
49 JVM_END
```

FIGURE 4.5 – Code source original de la fonction en charge de l'inspection de pile dans OpenJDK 1.7

La fonction *JVM_GetStackAccessControlContext* définie par la JVM a pour mission d'inspecter la pile d'exécution du thread courant et de retourner la liste des contextes de sécurité JAAS présents dans celle-ci. Les commentaires aux lignes 9 à 13 donnent l'algorithme général. La seconde condition au test de la ligne 36 exclut l'objet courant du contrôle s'il n'a pas été possible de trouver un contexte de sécurité pour cet objet.

autant ignorer les flux qui transitent par eux grâce à une règle particulière dans la politique.

L'idée est de remarquer qu'une règle de type *implicit allow* permettrait à n'importe quel objet de participer à tout type de relations sans pour autant être en contradiction avec d'éventuelles autres règles. En effet, au titre de la suggestion 4.1, une règle de type *implicit allow* sera toujours évaluée en dernier par rapport à une règle explicite ou bien un *implicit deny*. Nous verrons dans la section 4.3 sur le calcul automatisé des politiques comment utiliser au mieux cette idée qui est, selon nous, plus élégante que de soustraire brutalement des objets au contrôle d'accès comme le fait JAAS.

**Suggestion 4.2 (Suppression du bruit de contrôle)** *Soit un privilège $P_{all}(q)$ qui autorise tout objet à réaliser n'importe quelle relation. Pour tout objet du système, il existe une règle par défaut telle que :*

$$implicit\ allow\ P_{all}(q)$$

Continuons ainsi notre exemple avec les objets apprenti, maçon, petite fille et réfrigérateur. Nous avons déjà calculé précédemment les automates de contrôle relatifs à ce cas d'étude (cf. figure 4.4, page 94). L'objectif est maintenant de faire reconnaître à ces quatre automates la séquence d'interactions du début en suivant le principe des machines de Turing. La séquence où l'apprenti demande une bière au maçon qui en demande une à la petite fille qui la prend au réfrigérateur correspond à un programme pour nos automates.

$$o_a \xrightarrow{invoke} o_m \xrightarrow{invoke} o_x \xrightarrow{invoke} o_f \xrightarrow{invoke} o_y \xrightarrow{invoke} o_r$$

Ainsi, nous commençons par définir la politique initiale de manière à réduire le "bruit" dans le processus de décision selon la suggestion 4.2. Puis en appliquant l'algorithme 3 sur cette séquence en partant des appels les plus anciens vers les plus récents, nous obtenons les décisions suivantes :

**Préparation de l'inspection**

Il n'existe pas d'étape active pour l'instant :

$$\acute{E}tapes\ actives \equiv \emptyset$$

**Entrées** : Une séquence d'interactions représentant une pile d'appel;
**Données** : Soit $State(auto_x, q_{active}, o_{state})$ l'étape active $q_{active}$ d'un automate $auto_x$ activée par un objet $o_{state}$;
Soit *Policy* la politique utilisée lors de l'inspection de pile;
**Output** : Acceptation ou rejet de la séquence;

**pour chaque** *appel de méthode* $o_{source} \xrightarrow{invoke} o_{target}$ **faire**
  `// 1°) On recherche les automates à utiliser`
  **pour chaque** *automate* $auto_x$ **faire**
    **si** $Signature(o_{source})$ *représente une étape initiale* $q_{init}$ **alors**
      Créer l'étape active $State(auto_x, q_{init}, o_{source})$;
      Mettre à jour *Policy* avec les règles de $q_{init}$;
    **fin**
  **fin**

  `// 2°) On applique la politique de sécurité`
  **si** *Policy interdit* $o_{source} \xrightarrow{invoke} o_{target}$ **alors**
    **retourner** Rejet;
  **fin**

  `// 3°) On vérifie si il y a un changement d'étape`
  **pour chaque** *étape active* $State(auto_x, q_{active}, o_{state})$ **faire**
    **si** $Signature(o_{state}) \xrightarrow{invoke} Signature(o_{target})$ *représente une condition de transition de* $q_{active}$ *vers* $q_{next}$ **alors**
      `// On rend effectif le changement d'étape`
      Mettre à jour $State(auto_x, q_{active}, o_{state})$ telle que $q_{active} = q_{next}$ et $o_{state} = o_{target}$;
      `// On applique la décision de l'automate`
      **si** $q_{next}$ *est une étape finale* **alors**
        **si** $q_{next}$ *accepte la relation* **alors**
          Ne rien faire;
        **sinon si** $q_{next}$ *rejette la relation* **alors**
          **retourner** Rejet;
        **sinon**
          Appliquer la décision de l'étape $q_{next}$;
        **fin**
      **sinon**
        Mettre à jour *Policy* avec les règles de $q_{next}$;
      **fin**
    **fin**
  **fin**
**fin**

`// 4°) Aucun automate n'a rejeté la séquence`
**retourner** Accept;

**Algorithme 3 :** Inspection de pile par automate

La politique initiale est la suivante :

$$\begin{aligned} &Politique \equiv \{ \\ &\quad allow\ P_{all}(*) \\ &\} \end{aligned}$$

**Analyse de** $o_a \xrightarrow{invoke} o_m \equiv$ "*Apprenti*" $\xrightarrow{invoke}$ "***Maçon***"

Les automates 4.3 et 4.4 possèdent une étape initiale $q_4$ qui est *Apprenti* :

$$\begin{aligned} &\acute{E}tapes\ actives \equiv \{ \\ &\quad State(4.3, q_4, o_a) \\ &\quad State(4.4, q_4, o_a) \\ &\} \end{aligned}$$

Nous mettons la politique à jour en conséquence :

$$\begin{aligned} &Politique \equiv \{ \\ &\quad allow\ P_{all}(*) \\ &\quad deny\ P_1(q_4) \equiv deny\ \text{"}Apprenti\text{"} \Longrightarrow \text{"}Fille\text{"} \\ &\quad deny\ P_2(q_4) \equiv deny\ \text{"}Apprenti\text{"} \Longrightarrow \text{"}Réfrigérateur\text{"} \\ &\quad deny\ P_3(q_4) \equiv deny\ \text{"}Apprenti\text{"} \overset{bière}{\Longrightarrow} \text{"}Fille\text{"} \\ &\quad deny\ P_4(q_4) \equiv deny\ \text{"}Apprenti\text{"} \overset{bière}{\Longrightarrow} \text{"}Maçon\text{"} \\ &\quad allow\ P_5(q_4) \equiv allow\ \text{"}Apprenti\text{"} \xrightarrow{invoke} \text{"}Maçon\text{"} \\ &\quad allow\ P_6(q_4) \equiv allow\ \text{"}Apprenti\text{"} \longrightarrow \text{"}Réfrigérateur\text{"} \\ &\} \end{aligned}$$

La politique de sécurité autorise l'interaction "*Apprenti*" $\xrightarrow{invoke}$ "*Maçon*" donc on continue.

La relation "*Apprenti*" $\xrightarrow{invoke}$ "*Maçon*" que nous observons correspond à une condition de transition dans l'automate 4.3 :

$$State(4.3, q_4, o_a) = State(4.3, q_1, o_m)$$

Par cette transition, nous atteignons une étape finale et nous appliquons donc la décision de l'automate 4.3 :

$$\begin{aligned} &Décision \equiv \{ \\ &\quad Automate\ 4.3 Accepte \\ &\} \end{aligned}$$

**Analyse de** $o_m \xrightarrow{invoke} o_x \equiv$ "***Maçon***" $\xrightarrow{invoke}$ "*Objet*"

L'automate 4.1 possède une étape initiale $q_1$ qui est *Maçon* :

$$\begin{aligned}
&\textit{Étapes actives} \equiv \{\\
&\quad State(4.1, q_1, o_m)\\
&\quad State(4.3, q_1, o_m)\\
&\quad State(4.4, q_4, o_a)\\
&\}
\end{aligned}$$

Nous mettons la politique à jour en conséquence :

$$\begin{aligned}
&\textit{Politique} \equiv \{\\
&\quad allow\ P_{all}(*)\\
&\quad deny\ P_1(q_4) \equiv deny\ \text{"}Apprenti\text{"} \Longrightarrow \text{"}Fille\text{"}\\
&\quad deny\ P_2(q_4) \equiv deny\ \text{"}Apprenti\text{"} \Longrightarrow \text{"}Réfrigérateur\text{"}\\
&\quad deny\ P_3(q_4) \equiv deny\ \text{"}Apprenti\text{"} \xRightarrow{bière} \text{"}Fille\text{"}\\
&\quad deny\ P_4(q_4) \equiv deny\ \text{"}Apprenti\text{"} \xRightarrow{bière} \text{"}Maçon\text{"}\\
&\quad allow\ P_5(q_4) \equiv allow\ \text{"}Apprenti\text{"} \xrightarrow{invoke} \text{"}Maçon\text{"}\\
&\quad allow\ P_6(q_4) \equiv allow\ \text{"}Apprenti\text{"} \longrightarrow \text{"}Réfrigérateur\text{"}\\
&\\
&\quad allow\ P_1(q_1) \equiv allow\ \text{"}Maçon\text{"} \Longrightarrow \text{"}Fille\text{"}\\
&\quad deny\ P_2(q_1) \equiv deny\ \text{"}Maçon\text{"} \Longrightarrow \text{"}Réfrigérateur\text{"}\\
&\quad deny\ P_3(q_1) \equiv deny\ \text{"}Maçon\text{"} \xRightarrow{bière} \text{"}Fille\text{"}\\
&\quad deny\ P_4(q_1) \equiv deny\ \text{"}Maçon\text{"} \xRightarrow{bière} \text{"}Maçon\text{"}\\
&\quad deny\ P_5(q_1) \equiv deny\ \text{"}Maçon\text{"} \xrightarrow{invoke} \text{"}Maçon\text{"}\\
&\quad deny\ P_6(q_1) \equiv deny\ \text{"}Maçon\text{"} \longrightarrow \text{"}Réfrigérateur\text{"}\\
&\}
\end{aligned}$$

La politique de sécurité autorise l'interaction "*Maçon*" $\xrightarrow{invoke}$ "*Objet*" donc on continue (application de la règle par défaut). L'interaction que nous observons ne correspond à aucune condition de transition.

**Analyse de** $o_x \xrightarrow{invoke} o_f \equiv$ "*Objet*" $\xrightarrow{invoke}$ "*Fille*"

Aucun automate ne possède une étape initiale correspondant à *Objet*. Les étapes actives restent identiques :

$$\textit{Étapes actives} \equiv \{$$

$$
\begin{aligned}
&State(4.1, q_1, o_m)\\
&State(4.3, q_1, o_m)\\
&State(4.4, q_4, o_a)\\
\}&
\end{aligned}
$$

Il n'est pas non plus nécessaire de mettre à jour la politique. La politique de sécurité autorise l'interaction "*Maçon*" $\xrightarrow{invoke}$ "*Fille*" car il s'agit d'un flux d'information "*Maçon*" $\Longrightarrow$ "*Fille*" donc on continue.

La relation "*Maçon*" $\Longrightarrow$ "*Fille*" que nous observons correspond à une condition de transition dans l'automate 4.1 :

$$State(4.1, q_1, o_m) = State(4.1, q_2, o_f)$$

Par cette transition, nous atteignons une étape non finale et nous mettons à jour la politique en conséquence :

$$
\begin{aligned}
Politique \equiv \{&\\
&allow\ P_{all}(*)\\
&deny\ P_1(q_4) \equiv deny\ \text{"}Apprenti\text{"} \Longrightarrow \text{"}Fille\text{"}\\
&deny\ P_2(q_4) \equiv deny\ \text{"}Apprenti\text{"} \Longrightarrow \text{"}Réfrigérateur\text{"}\\
&deny\ P_3(q_4) \equiv deny\ \text{"}Apprenti\text{"} \overset{bière}{\Longrightarrow} \text{"}Fille\text{"}\\
&deny\ P_4(q_4) \equiv deny\ \text{"}Apprenti\text{"} \overset{bière}{\Longrightarrow} \text{"}Maçon\text{"}\\
&allow\ P_5(q_4) \equiv allow\ \text{"}Apprenti\text{"} \xrightarrow{invoke} \text{"}Maçon\text{"}\\
&allow\ P_6(q_4) \equiv allow\ \text{"}Apprenti\text{"} \longrightarrow \text{"}Réfrigérateur\text{"}\\
&\\
&allow\ P_1(q_1) \equiv allow\ \text{"}Maçon\text{"} \Longrightarrow \text{"}Fille\text{"}\\
&deny\ P_2(q_1) \equiv deny\ \text{"}Maçon\text{"} \Longrightarrow \text{"}Réfrigérateur\text{"}\\
&deny\ P_3(q_1) \equiv deny\ \text{"}Maçon\text{"} \overset{bière}{\Longrightarrow} \text{"}Fille\text{"}\\
&deny\ P_4(q_1) \equiv deny\ \text{"}Maçon\text{"} \overset{bière}{\Longrightarrow} \text{"}Maçon\text{"}\\
&deny\ P_5(q_1) \equiv deny\ \text{"}Maçon\text{"} \xrightarrow{invoke} \text{"}Maçon\text{"}\\
&deny\ P_6(q_1) \equiv deny\ \text{"}Maçon\text{"} \longrightarrow \text{"}Réfrigérateur\text{"}\\
&\\
&deny\ P_1(q_2) \equiv deny\ \text{"}Fille\text{"} \Longrightarrow \text{"}Fille\text{"}\\
&allow\ P_2(q_2) \equiv allow\ \text{"}Fille\text{"} \Longrightarrow \text{"}Réfrigérateur\text{"}\\
&deny\ P_3(q_2) \equiv deny\ \text{"}Fille\text{"} \overset{bière}{\Longrightarrow} \text{"}Fille\text{"}\\
&deny\ P_4(q_2) \equiv deny\ \text{"}Fille\text{"} \overset{bière}{\Longrightarrow} \text{"}Maçon\text{"}
\end{aligned}
$$

$$
\begin{aligned}
&deny\ P_5(q_2) \equiv deny\ "Fille" \xrightarrow{invoke} "Maçon"\\
&deny\ P_6(q_2) \equiv deny\ "Fille" \longrightarrow "Réfrigérateur"\\
\}
\end{aligned}
$$

**Analyse de** $o_f \xrightarrow{invoke} o_y \equiv "Fille" \xrightarrow{invoke} "Objet"$

Aucun automate ne possède une étape initiale correspondant à *Fille*. Les étapes actives restent identiques :

$$
\begin{aligned}
&\acute{E}tapes\ actives \equiv \{\\
&\quad State(4.1, q_2, o_f)\\
&\quad State(4.3, q_1, o_m)\\
&\quad State(4.4, q_4, o_a)\\
\}
\end{aligned}
$$

Il n'est pas non plus nécessaire de mettre à jour la politique. La politique de sécurité autorise l'interaction $"Fille" \xrightarrow{invoke} "Objet"$ donc on continue (application de la règle par défaut). L'interaction que nous observons ne correspond à aucune condition de transition.

**Analyse de** $o_y \xrightarrow{invoke} o_r \equiv "Objet" \xrightarrow{invoke} "\boldsymbol{Réfrigérateur}"$

Aucun automate ne possède une étape initiale correspondant à *Objet*. Les étapes actives restent identiques :

$$
\begin{aligned}
&\acute{E}tapes\ actives \equiv \{\\
&\quad State(4.1, q_2, o_f)\\
&\quad State(4.3, q_1, o_m)\\
&\quad State(4.4, q_4, o_a)\\
\}
\end{aligned}
$$

Il n'est pas non plus nécessaire de mettre à jour la politique. La politique de sécurité autorise l'interaction $"Fille" \xrightarrow{invoke} "Réfrigérateur"$ car il s'agit d'un flux d'information $"Fille" \Longrightarrow "Réfrigérateur"$ et d'une interaction $"Apprenti" \longrightarrow "Réfrigérateur"$.

Les relations "*Fille*" $\Longrightarrow$ "*Réfrigérateur*" et "*Apprenti*" $\longrightarrow$ "*Réfrigérateur*" que nous observons correspondent à une condition de transition dans les automates 4.1 et 4.4 :

$$State(4.1, q_2, o_f) = State(4.1, q_3, o_r)$$
$$State(4.4, q_4, o_a) = State(4.4, q_3, o_r)$$

Par ces transitions, nous atteignons des étapes finales et nous appliquons donc la décision des automates 4.1 et 4.4 :

$$\begin{aligned} &Décision \equiv \{ \\ &\quad Automate\ 4.3 Accepte \\ &\quad Automate\ 4.1 Accepte \\ &\quad Automate\ 4.4 Rejette \\ &\} \end{aligned}$$

**Décision finale et interprétation des résultats**

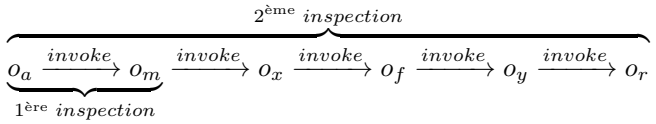
Il n'y a plus d'interactions à observer par inspection de pile. Sur nos quatre automates, trois ont reconnus la relation qu'ils contrôlent et seul l'automate 4.4 a rejeté la séquence observée. Cela signifie qu'il y a donc violation de la politique du cas d'étude et en conséquence nous devons interrompre la dernière interaction de cette séquence. Ce résultat est conforme à ce que l'on attendait car l'automate 4.4 a pour rôle de justement détecter un accès illégitime de l'apprenti au réfrigérateur, accès que JAAS n'avait pas été en mesure de détecter (cf. section 4.1.2). La raison vient de la conjugaison de trois éléments :

1. Contrairement à JAAS, nous n'arrêtons pas l'inspection au premier objet privilégié que l'on rencontre. De fait nous analysons donc l'intégralité de la pile d'exécution ce qui permet à l'automate 4.4 d'identifier l'élévation de privilège de l'apprenti sur le réfrigérateur.

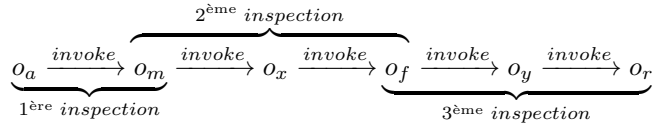
2. La règle 4.1 exprime une élévation de privilèges de l'objet *Maçon*, chose que JAAS ne sait pas exprimer correctement. Pour que l'apprenti puisse effectivement accéder au réfrigérateur il aurait fallu exprimer une élévation de privilèges sur le même format que pour le maçon. Ce qui n'est pas le cas ici, d'où le rejet par l'automate 4.4.

3. Dans l'approche JAAS, chaque objet privilégié du système déclenche une inspection de pile. Or, avec notre approche par automate, seuls les objets dont la signature correspond à un état final déclenchent une inspection, c'est à dire par les objets *Maçon* et *Réfrigérateuur*. Nous n'avons présenté

que la seconde inspection alors qu'une première inspection avait déjà été acceptée par l'automate 4.3 et dont la décision est confirmée par la seconde inspection. Ainsi, d'un point de vue qualitatif, nous obtenons donc un meilleur résultat que JAAS mais avec une inspection de pile en moins (cf. figure 4.6).




(a) Inspections de pile réalisées par automates




(b) Inspections de pile réalisées par JAAS

FIGURE 4.6 – Comparaison entre JAAS et les automates

### 4.1.5 Discussion sur le cas d'étude

Par ce cas d'étude nous voulions mettre en lumière les forces et les faiblesses de JAAS. Les scénarios du cas d'étude d'origine dont nous nous sommes inspirés [83] sont très biens pour mettre en lumière la logique derrière JAAS. L'idée de mettre le focus sur uniquement les objets pertinents à contrôler est selon nous une bonne stratégie car elle permet de concentrer le contrôle sur ce qui est essentiel. C'est à dire quels sont les objets concernés et les relations entre eux ; tout autre objet étant alors des intermédiaires participant à ces relations.

Par notre implémentation du cas d'étude original située en annexe, nous attestons que JAAS fait exactement ce que l'on attend de lui lorsqu'il y a une élévation directe des privilèges (via un seul objet privilégié). Mais quand nous avons introduit un quatrième objet de manière à tester JAAS dans le cas d'une élévation indirecte (via plusieurs objets privilégiés) nous avons pu observer que JAAS était totalement inefficace. Notre analyse nous a conduit à penser que le fait de réaliser une inspection de pile par segments plutôt que globalement, ajoutée au fait que les élévations de privilèges sont réalisées par du code à des points fixes du programme, réduit fortement les capacités de JAAS à observer correctement les relations entre objets.

Notre idée d'utiliser des automates repose justement sur cette nécessité d'observer avec exactitude les relations entre objets. Et en ce sens, nous allons donc au-delà des capacités de JAAS. En terme d'implémentation, nous avons fait le choix de réutiliser le principe de l'inspection de pile mais en imposant une analyse globale. C'est l'inspection de pile qui sert à observer les relations et permet

de faire avancer les automates. Un véritable moniteur de référence serait cependant plus adapté mais, comme nous le verrons dans le chapitre suivant, celui-ci introduit une très grande compléxité tant de mise en œuvre que du point de vue des politiques. Ainsi, ce que nous proposons n'est pas une simple amélioration de JAAS mais plutôt une alternative à celui-ci qui réutilise ses points forts et laisse de coté ses limitations intrinsèques.

La question à se poser maintenant est de savoir comment se comporte notre logique par automates face au modèle d'attaques que nous avons établi pour Java. Nous apporterons cette réponse dans la section suivante.

## 4.2 Évaluation du modèle d'attaques

Fondamentalement Java est un système à objets basé sur les classes. Il est constitué d'un hyperviseur (JRE) et d'une ou plusieurs machines virtuelles (JVM) en charge d'exécuter les programmes Java compilés. Nous savons que le modèle général que nous avons est suffisant pour caractériser les relations entre objets d'un système à classes et par la même d'établir des règles de contrôle liées à chacune de ces relations [80, 81]. Nous proposons d'analyser ici l'exploitation de vulnérabilités Java connues afin de déterminer si celles-ci sont observables et contrôlables par notre logique basée sur les automates. Pour cela, nous nous placerons d'abord dans un cas général puis dans le cas d'un véritable malware.

### 4.2.1 Corruption de mémoire

Les attaques par corruption de mémoire consistent à modifier de façon illégitime les données stockées en mémoire. Très commune dans les environnements d'exécution natifs comme C/C++, on y retrouve les vulnérabilités de type débordement de tampon ainsi que les chaînes de formatage mal formées. L'idée est d'obtenir le contrôle du système en altérant les données de l'environnement. Ainsi, ce type d'attaque peut avoir une incidence sur le système à objets sous réserve que les données modifiées soient celles d'un objet. En effet, nous excluons ici les cas où l'attaque par corruption de mémoire vise exclusivement le noyau du système à objets comme l'hyperviseur Java.

#### Cas général

En pratique, cette attaque se traduit dans le système par l'apparition d'un objet compromis voir malveillant (cf. figure 4.7). Nous pourrions envisager de chercher

à détecter de tels objets parmi les objets sains du système mais nous nous heurterions au problème des généraux byzantins [84]. Ce problème suppose un ensemble de généraux (ici nos objets) qui doivent se coordonner pour lancer une attaque sur l'ennemi (ici remplir les fonctionnalités du système). L'objectif est de distinguer parmi eux lesquels sont loyaux (c-à-d les objets sains) de ceux qui agissent pour l'ennemi (c-à-d ceux qui ont été corrompus). Or il a été montré [85] qu'un tel problème est indécidable, c'est à dire qu'il n'existe aucun algorithme pour le résoudre. Cela signifie donc qu'il ne sera pas possible d'empêcher la réalisation de cette attaque.

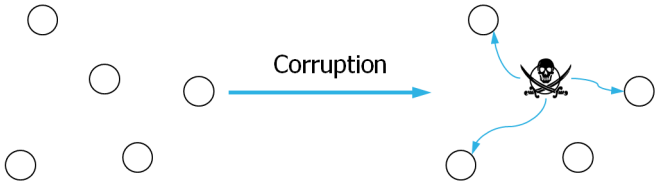



FIGURE 4.7 – Représentation d'une attaque par corruption de mémoire.

Par contre, nous pourrions envisager de restreindre les privilèges des objets du système et ainsi limiter les actions d'un objet compromis. En effet, un objet dont les données ont été corrompues peut adopter un comportement illégitime voir malveillant. Par exemple, la corruption de mémoire peut affecter les données exécutables d'une méthode de cet objet induisant ainsi des embranchements de code non-prévus et, par conséquent, des interactions anormales entre objets. Or notre modélisation et notre logique de contrôle par automate sont justement conçues pour formaliser les relations observables entre objets, qu'elles soient légitimes ou non, et ainsi appliquer une politique de contrôle d'accès. Dès lors, peu importe si les objets que l'on contrôle sont sains ou compromis vu que nous sommes en mesure de les forcer à respecter les objectifs de sécurité établis dans la politique.

### Cas Java

Il est possible de trouver des rapports de vulnérabilité (CVE) relatives à des attaques par corruption de mémoire pour les différentes implémentations de Java. Mais la grande majorité d'entre elles concernent l'hyperviseur et non la mémoire du système à objets. Par contre, nous pouvons traiter une technique d'instrumentation qui consiste à injecter du code dans l'image en mémoire des programmes en cours d'exécution ; nous pouvons notamment citer la bibliothèque *Microsoft Détours*[4] [86] pour les applications natives. Une technique comparable est disponible pour les environnements Java dont le principe est d'utiliser les API de réflexion afin de modifier les classes chargées en mémoire. L'injection de données peut être légitime comme dans le cas d'un profilage d'application avec *hprof*[5]. Mais cela pourrait également permettre à un hébergeur de services

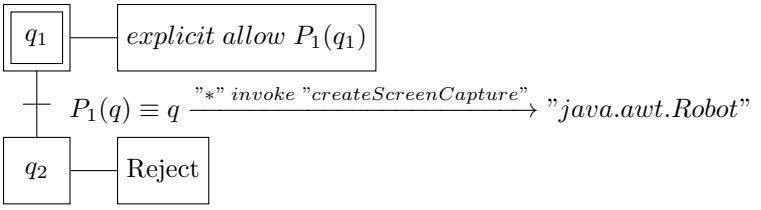
4. `http://research.microsoft.com/en-us/projects/detours/`
5. `http://docs.oracle.com/javase/8/docs/technotes/samples/hprof.html`

web un peu trop curieux d'injecter du code espion dans les applications qu'il héberge et ce à l'insu de ses clients.

C'est pour cette raison que nous proposons d'étudier le cas d'un *screen-logger* Java qui serait injecté dans une application légitime. Il s'agit d'une variante du *key-logger* dont le principe est de capturer à intervalles réguliers ce qui est affiché sur l'écran standard de l'utilisateur. Ce type de malware est très facile à coder car l'API Java fournit toutes les fonctionnalités nécessaires et, au pire, il est simple de trouver des composants logiciels clefs en main comme par exemple ASL [6] pour Android.

Dans le cas de Java, il suffit ainsi de créer une instance de la classe *Robot* [7] puis d'appeler la méthode *createScreenCapture*. Comme précisé pour le cas général, nous ne pouvons empêcher la corruption de mémoire. Par contre nous pouvons restreindre l'accès à la classe *Robot* ainsi qu'à son membre *createScreenCapture*, c'est à dire rejeter toute interaction à destination de ce type d'objet et de ce membre. Ce type de contrainte est facile à transcrire en un automate de contrôle :

$$Q \equiv \{q_1 = "*",\ q_2 = "java.awt.Robot"\}$$




Ainsi, dans le cas où un objet du système tente de prendre une capture d'écran de l'utilisateur, cet automate sera en mesure de le détecter et de l'interdire en conséquence. Il est intéressant de remarquer que JAAS définit des permissions particulières [20] pour instancier un objet de la classe *Robot* et appeler la méthode *createScreenCapture* (*createRobot* et *readDisplayPixels* respectivement). Donc sans nous substituer à JAAS nous pouvons réutiliser ces permissions pour définir les règles élémentaires telles que $P_1(q)$ et les utiliser comme transitions dans nos automates (cf. tableau 4.1).

### 4.2.2 Confusion de types

L'attaque par confusion de types consiste à usurper le type d'un objet afin d'obtenir ses privilèges. Dans les faits il s'agit d'une opération de transtypage

6. `https://code.google.com/p/android-screenshot-library/`
7. `http://docs.oracle.com/javase/7/docs/api/java/awt/Robot.html`

| Classe | Méthode |
|---|---|
| java.awt.Robot | <init>()V |
| java.awt.Robot | <init>(java.awt.GraphicsDevice)V |
| java.awt.Graphics2d | setComposite(java.awt.Composite)V |

(a) Méthodes protégées selon [20]

$$P_1(q) = q \xrightarrow{\text{"}*\text{" invoke "}<init>()V\text{"}} \text{"}java.awt.Robot\text{"},$$

$$P_2(q) = q \xrightarrow{\text{"}*\text{" invoke "}<init>(java.awt.GraphicsDevice)V\text{"}} \text{"}java.awt.Robot\text{"},$$

$$P_3(q) = q \xrightarrow{\text{"}*\text{" invoke "}setComposite(java.awt.Composite)V\text{"}} \text{"}java.awt.Graphics2d\text{"}$$

(b) Transposition des capacités en relations élémentaires

TABLE 4.1 – Réécriture des capacités JAAS *(AWTPermission, createRobot)* et *(AWTPermission, readDisplayPixels)*

(cast) sur le type de l'objet directement, et non sur une de ses références. Cela conduit alors à une élévation de privilèges pour l'objet usurpateur. Celui-ci gagnant alors les privilèges de l'objet dont il usurpe le type. Il s'agit d'un type d'attaque relativement rare dans les systèmes à objets actuels car elle impacte directement la cohérence de tout le système. Cependant, même en étant peu probable, ce type d'attaque reste possible.

**Cas général**

Un système à objets ne peut en pratique vérifier le type d'un objet qu'au seul moment où l'attention de cet objet est requise. Par exemple, si un objet $o_1$ désire appeler une méthode d'un objet $o_2$, les types de ces deux objets ne seront vérifiés qu'au moment de l'appel et non avant. L'explication provient du résultat d'impossibilité du théorème de Rice [30] qui implique que le système ne peut analyser statiquement, donc à l'avance, les flots du système. En conséquence, un objet peut abuser le système en modifiant ses types.

Prenons l'exemple d'un système constitué de trois objets et supposons une règle de la forme *allow* "$A$" $\longrightarrow$ "$B$" qui autorise seulement les objets de type $A$ à interagir avec les objets de type $B$. Ainsi, ce système fait apparaître que seul l'objet $o_A$ pourra accéder à l'objet $o_B$ car, implicitement, la politique considérée

n'autorise pas les objets du type de $o_C$ à accéder aux objets du type de $o_B$ :

$$O \equiv \{o_A, o_B, o_C\}$$
$$Types(o_A) \equiv "A" \quad Types(o_B) \equiv "B" \quad Types(o_C) \equiv "C"$$

Toute interaction $o_A \longrightarrow o_B$ est explicitement autorisée.
Toute interaction $o_C \longrightarrow o_B$ est implicitement interdite.

Il s'agit ici d'un résultat conforme à ce qu'exprime la politique considérée. Mais supposons un défaut du système qui permettrait à l'objet $o_C$ de se maquiller en un objet de type $A$. Pour qu'un objet usurpe le type d'un autre, ce premier objet doit au préalable exister dans le système et donc nous connaissons déjà son type initial. Or et grâce à notre modélisation des systèmes à objets, nous savons qu'un objet peut posséder plusieurs types et, plus particulièrement, le fait d'usurper le type d'un autre objet ne fait que lui en rajouter un nouveau. Ainsi le type calculé de l'objet $o_C$ sera à la fois son type d'origine qui est "C" et le type "A" qu'il a usurpé, c'est à dire : $Types(o_C) \equiv \{"A", "C"\}$.

Déjà nous pouvons remarquer que contrairement à l'attaque par corruption de mémoire, nous sommes en mesure de détecter la confusion de type. En effet, si nous nous plaçons dans un système à classes comme Java, nous savons que les types d'un objet sont toujours égaux aux noms de sa classe (cf. notation 3.12). Or, cette égalité n'est plus vérifiée lorsqu'il y a une confusion de type d'où, selon nous, l'importance de conserver les types d'origines de chaque objet et, surtout, de les calculer le plus tôt possible. Nous pouvons d'ailleurs étendre cette recommandation à la signature d'objet pour limiter les risques de confusion de type dans les autres familles de système à objets.

**Suggestion 4.3 (Conservation des types)** *La signature de chaque objet doit être calculée au moment de sa construction puis conservée tout au long de la vie de l'objet.*

Plus encore, si nous développons la politique considérée, nous savons qu'il existe une règle implicite de type *deny* interdisant aux objets de type "C" d'accéder aux objets de type "B", et ce au titre du principe de *tout ce qui n'est pas explicitement autorisé est interdit* :

$$explicit\ allow\ "A" \longrightarrow "B"$$
$$implicit\ deny\ "C" \longrightarrow "B"$$

Ainsi, lorsque nous observerons une interaction $o_C \longrightarrow o_B$ la première autorisera l'accès car l'objet $o_3$ possède le type $A$. Par contre la seconde règle l'interdira

car l'objet $o_C$ possède aussi le type $C$ qui est son type d'origine. Donc sans règle explicite de type *allow*, l'objet $o_C$ ne pourra pas accéder à l'objet $o_B$ et ce malgré la réussite apparente de l'attaque par confusion de type. Cela renforce donc d'autant plus la nécessité de conserver les types des objets du système.

**Cas Java**

Il est difficile de donner un code source Java qui illustre cette attaque car, comme expliqué dans l'état de l'art, ce type de vulnérabilité est intimement lié à une implémentation particulière de l'hyperviseur. C'est pourquoi nous allons plutôt analyser un exploit concret relatif à ce type d'attaque.

Dans [87], Michael Schierl propose une analyse de la vulnérabilité CVE-2012-1723. Il s'agit d'une vulnérabilité de OpenJDK pour laquelle une optimisation de l'analyseur de bytecode induit une vérification partielle des instructions d'accès aux champs d'un objet. Malgré la beauté technique de cette attaque, nous ne présenterons pas tous les détails de l'exploit proposé. Nous retiendrons juste qu'il s'appuie sur une condition de concurrence entre l'analyseur de bytecodes et la mise à jour d'une référence afin d'usurper un champ privé pour un champ de visibilité publique.

Le principe de l'exploit est de définir une classe arbitraire, ici *ConfusedCL*, qui a pour parent le type *Classloader*. Puis, par confusion de type, l'exploit maquille le type de son chargeur de classes par celui de cette classe héritée. L'objectif est de pouvoir appeler la méthode *defineClass* de visibilité protégée par son homologue *define* de même type dans la classe héritée mais de visibilité publique. Si l'attaque réussit alors le malware pourra ordonner à son chargeur de classe le chargement de n'importe quelle autre classe.

Ainsi, nous nous retrouvons dans le même cas de figure que pour le cas général. En effet, le chargeur de classes système, que l'on identifiera par l'objet $o_S$, est initialement de type *Classloader*. Puis, à un moment de l'exécution du malware, cet objet $o_S$ obtient un type supplémentaire qui *ConfusedCL*. Le programme de la figure 4.8 est adapté de l'exploit original que nous pouvons facilement

```java
public class Attacker extends Applet {

  public void init() {
    super.init();

    // [...]

    try {
      ConfusedClassLoader cl = Confuser.confuse(getClass().
          getClassLoader());
      String[] names = { "msf.x.PayloadX", "msf.x.
          PayloadX$StreamConnector" };
      String[] paths = { "/msf/x/PayloadX.class", "/msf/x/
          PayloadX$StreamConnector.class" };

      String port = getParameter("lport");
      ConfusedClassLoader.define(cl, names, new byte[][] {
          loadClass(paths[0]), loadClass(paths[1]) },
          getParameter("data"), getParameter("jar"),
          getParameter("lhost"), port == null ? 4444 :
          Integer.parseInt(port));
    } catch (Exception e) { e.printStackTrace(); }
  }

  // [...]
}
```

(a) Classe principale de l'exploit

```java
public class ConfusedClassLoader extends ClassLoader {

  public static void define(ConfusedClassLoader cl, String[]
      name, byte[][] data, String hexdata, String jar, String
       lhost, int lport) {
    try {
      Permissions p = new Permissions();
      p.add(new AllPermission());
      ProtectionDomain pd = new ProtectionDomain(new
          CodeSource(null, new Certificate[0]), p);

      Class clazz = cl.defineClass(name[0], data[0], 0, data
          [0].length, pd);
      cl.defineClass(name[1], data[1], 0, data[1].length, pd)
          ;

      // [...]

      clazz.newInstance();
    } catch (Exception e) { e.printStackTrace(); }
  }

  // [...]
}
```

(b) Classe usurpatrice du type *ClassLoader*

FIGURE 4.8 – Exploit de la CVE-2012-1723

formaliser avec notre modèle :

$$O \equiv \{\cdots, o_5, o_S, o_7, \cdots\}, \quad C \equiv \{c_1, c_2, c_3, c_4\}, \quad c_4 \twoheadrightarrow c_2,$$

$$\begin{aligned}
&Noms(c_1) \equiv \text{"}Confuser\text{"}, \quad Méthodes(c_1) \equiv \{f_1^1\}\\
&Noms(c_2) \equiv \text{"}Classloader\text{"}, \quad Méthodes(c_2) \equiv \{\cdots, f_7^2, \cdots\}\\
&Noms(c_3) \equiv \{\text{"}Attacker\text{"}, \text{"}Applet\text{"}\}, \quad Méthodes(c_3) \equiv \{f_1^3, \cdots\}\\
&Noms(c_4) \equiv \{\text{"}ConfusedClassLoader\text{"}, \text{"}ClassLoader\text{"}\}, \quad Méthodes(c_4) \equiv \{f_1^4\}
\end{aligned}$$

$$\begin{aligned}
&Noms(f_1^1) \equiv \text{"}confuse\text{"}, \quad Types(f_1^1) \equiv \{\text{"}exécutable\text{"}, \cdots\}\\
&Noms(f_7^2) \equiv \text{"}defineClass\text{"}, \quad Types(f_7^2) \equiv \{\text{"}exécutable\text{"}, \cdots\}\\
&Noms(f_1^3) \equiv \text{"}init\text{"}, \quad Types(f_1^3) \equiv \{\text{"}exécutable\text{"}, \cdots\}\\
&Noms(f_1^4) \equiv \text{"}define\text{"}, \quad Types(f_1^4) \equiv \{\text{"}exécutable\text{"}, \text{"}ConfusedClassLoader\text{"}, \cdots\}
\end{aligned}$$

$$\begin{aligned}
Classes(o_5) \equiv c_1, \quad Types(o_5) \equiv Noms(c_1) \equiv \text{"}Confuser\text{"}\\
Classes(o_S) \equiv c_2, \quad Types(o_S) \equiv Noms(c_2) \equiv \text{"}Classloader\text{"}\\
Classes(o_7) \equiv c_3, \quad Types(o_7) \equiv Noms(c_3) \equiv \{\text{"}Attacker\text{"}, \text{"}Applet\text{"}\}
\end{aligned}$$

À l'exécution, nous pouvons observer un premier appel à la classe malicieuse *Confuser* :

$$\begin{aligned}
&o_7 \xrightarrow{c_3::f_1 \ invoke \ c_1::f_1} o_5\\
\equiv &\text{"}Attacker\text{"} \xrightarrow{\text{"}init\text{"} \ invoke \ \text{"}confuse\text{"}} \text{"}Confuser\text{"}
\end{aligned}$$

$$\begin{aligned}
&o_5 \xrightarrow{c_1::f_1 \ return \ c_3::f_1} o_7\\
\equiv &\text{"}Confuser\text{"} \xrightarrow{\text{"}confuse\text{"} \ return \ \text{"}init\text{"}} \text{"}Attacker\text{"}
\end{aligned}$$

Plus particulièrement, si l'on compare les flux de données entre l'appel à la méthode *confuse* de l'objet $o_5$ de type *Confuser* et le retour de cet appel, nous pouvons apprécier que le type de l'objet $o_S$ passé en paramètre est transtypé de façon illégalle :

$$o_7 \overset{o_S}{\Longrightarrow} o_5 \equiv \text{"}Attacker\text{"} \xRightarrow{\{\text{"}Classloader\text{"}\}} \text{"}Confuser\text{"}$$

$$o_5 \overset{o_S}{\Longrightarrow} o_7 \equiv \text{"}Confuser\text{"} \xRightarrow{\{\text{"}Classloader\text{"},\text{"}ConfusedClassLoader\text{"}\}} \text{"}Attacker\text{"}$$

L'observation de ce transtypage est suffisant pour décider de la non-légitimité de la première interaction. Il s'agit d'ailleurs de la stratégie suivie par le vérificateur de code de la JVM mais qui, ici, a néanmoins été mis en échec du fait de la vulnérabilité CVE-2012-1723. Sans un mécanisme de protection adéquat, le malware est libre de continuer son exécution en injectant des classes malicieuses dans le noyau de la JVM :

$$so_7 \xrightarrow{c_3::f_1 \ invoke \ c_4::f_1} c_4$$
$$\equiv \text{"}Attacker\text{"} \xrightarrow{\text{"}init\text{"} \ invoke \ \text{"}define\text{"}} \text{"}ConfusedClassLoader\text{"}$$

$$c_4 \xrightarrow{c_4::f_1 \ invoke \ c_2::f_7} o_S$$
$$\equiv \text{"}ConfusedClassLoader\text{"} \xrightarrow{\text{"}define\text{"} \ invoke \ \text{"}defineClass\text{"}} \text{"}Classloader\text{"}$$

Ainsi, notre modèle général nous permet d'observer et donc contrôler les attaques par confusion de types dans Java. Remarquons toutefois que cet exploit Java s'exécute en tant qu'applet, c'est à dire sans privilèges particuliers (cf. relation d'héritage de la classe *Attacker*). L'objectif de cet exploit est donc de s'échapper de la sandbox Java et ainsi obtenir un contrôle total de la JVM. Traditionnellement, la technique consiste à prendre le contrôle d'un classloader, comme c'est le cas ici, dans le but d'injecter des classes malicieuses dans le noyau de la JVM.

Or, selon les spécifications de sécurité de JAAS [20], la création d'un classloader nécessite une permission particulière qu'une applet Java ne possède pas (permission *createClassloader*). Nous savons que l'exploit présenté ici arrive à abuser JAAS en se servant de son classloader comme classe privilégiée intermédiaire (cf. conclusions de la section 4.1). Mais il est intéressant de noter que comme dans le cas de l'attaque par corruption de mémoire, nous identifions là encore un privilège Java particulier qui limite l'accès à un objet privilégié de la JVM.

Au final, nous pouvons là encore réutiliser une permission JAAS pour définir des règles élémentaires utilisables comme transitions dans nos automates (cf. tableau 4.3).

### 4.2.3 Défaut de contrôle d'accès

La littérature montre que le moniteur de référence de Anderson [21] est un prérequis nécessaire à la garantie de politiques de sécurité. Celui-ci se définit en trois critères dont le premier est qu'il doit absolument intercepter toutes les demandes d'accès ; les deux autres étant qu'il ne doit pas être influençable et

| Classe | Méthode |
|---|---|
| java.lang.ClassLoader | <init>()V |
| java.lang.ClassLoader | <init>(java.lang.ClassLoader)V |
| java.net.URLClassLoader | <init>(*)V |
| java.security.SecureClassLoader | <init>(*)V |

(a) Méthodes protégées selon [20]

$$P_4(q) = q \xrightarrow{\text{"*" } invoke \text{ "<init>()V"}} \text{"}java.lang.ClassLoader\text{"},$$

$$P_5(q) = q \xrightarrow{\text{"*" } invoke \text{ "<init>(java.lang.ClassLoader)V"}} \text{"}java.lang.ClassLoader\text{"},$$

$$P_6(q) = q \xrightarrow{\text{"*" } invoke \text{ "<init>(*)V"}} \text{"}java.net.URLClassLoader\text{"},$$

$$P_7(q) = q \xrightarrow{\text{"*" } invoke \text{ "<init>(*)V"}} \text{"}java.security.SecureClassLoader\text{"}$$

(b) Transposition de la capacité en relations élémentaires

TABLE 4.3 – Réécriture de la capacité JAAS *(RuntimePermission, createClassloader)*

que son implémentation soit formellement prouvée par rapport au modèle de protection.

Ce que l'on nomme "défaut de contrôle d'accès" correspond à la situation où le moniteur de référence est soit contourné, soit inutile lorsque les privilèges vérifiés ne sont pas corrects. Bien évidemment nous excluons le cas trivial où il n'existe tout simplement pas de contrôle d'accès. C'est pourquoi nous allons surtout traiter l'aspect d'un contrôle d'accès qui ne serait pas appliqué correctement.

### Cas général

De façon générale, il existe deux stratégies pour abuser un moniteur de référence. Soit l'attaquant utilise un canal caché pour rendre les interactions malicieuses imperceptibles, donc non contrôlables, comme dans le cas de la CVE-2006-2769 réputée pour avoir mis à mal le détecteur d'intrusion Snort. Soit à utiliser des séquences d'interactions qui, par une erreur de politique ou d'optimisation, échappent au contrôle du modèle de protection. Le second cas est à différencier du premier sur le fait que les interactions restent observables bien qu'elles ne soient pas observées.

La difficulté est que notre approche se base exclusivement sur les relations observables (cf. hypothèse 3.1). Ce qui implique que nous avons besoin d'un véritable

moniteur de référence pour la mettre en œuvre. En effet, seul un moniteur de référence qui remplit les trois critères de Anderson apporte les garanties nécessaires à l'exactitude des relations que l'on observe.

**Suggestion 4.4 (Moniteur de référence pour système à objets)**
*La mise en application des propositions 3.1, 3.2, 3.4, 3.5 et 3.3 sur le contrôle des relations entre objets nécessite un moniteur de référence de Anderson [21].*

Ainsi, si les critères du moniteur de référence sont bien respectés, une décision sera prise pour chaque demande d'accès et cette décision sera conforme à ce qui aura été observé : il ne peut y avoir de contournement du modèle de protection. Par contre, il est possible que la décision qui est prise ne respecte pas le principe de moindre privilèges [19]. C'est à dire qu'une erreur dans la politique de sécurité permet à un objet non privilégié d'accéder à un objet privilégié du système alors même que cela ne soit pas formalisé par un objectif de sécurité.

Nous allons donc supposer deux objets $o_p$ et $o_{np}$. La question est : peut on exprimer avec notre modèle général un objectif de sécurité qui autorise ou interdit un objet non-privilégié $o_{np}$ d'accéder à un objet privilégié $o_p$ ? Deuxième question : sommes nous en mesure d'appliquer cet objectif de sécurité ?

La réponse à la première question est triviale car notre modèle général des systèmes à objets nous permet d'exprimer une relation directe entre $o_p$ et $o_{np}$ telle que $o_p \longrightarrow o_{np}$ mais aussi une relation indirecte qui utilise un objet intermédiaire $o_i$ telle que $o_p \longrightarrow o_i \longrightarrow o_{np}$. Nous avons d'ailleurs vu dans le cas d'étude sur le maçon, la petite fille et le réfrigérateur que nous étions en mesure de contrôler ce type de relation à l'aide d'automates. Nous sommes donc en mesure de contrôler les relations entre un objet non-privilégié et un objet privilégié et de respecter le principe de moindre privilèges.

### Cas Java

Le défaut de contrôle d'accès est un problème récurrent dans Java qui est à distinguer de l'abus de l'inspection de pile avec lequel il est intimement lié. Parce que JAAS s'appuie principalement sur du code inséré dans le programme dont le but est d'appliquer les règles de contrôle (cf. section 2.3.4 de l'état de l'art), il existe une probabilité non négligeable que le développeur ait fait une erreur d'implémentation.

Un exemple parlant est celui de la vulnérabilité CVE-2008-5353 dont nous pouvons trouver une analyse dans [88]. Il s'agit certes d'une vulnérabilité ancienne

mais elle reste populaire parmi les pentesteurs [8] car elle a la particularité d'avoir un taux d'exploitabilité de 100% et d'être multi-platformes. Pour information, l'éditeur d'antivirus Kaspersky indique des tentatives d'exploitation de cette vulnérabilité dans ses statistiques d'avril 2012 [9].

```
1  // If there's a ZoneInfo object, use it for zone.
2  try {
3    ZoneInfo zi = (ZoneInfo)AccessController.doPrivileged(
4      new PrivilegedExceptionAction() {
5        public Object run() throws Exception {
6          return input.readObject();
7        }
8      });
9    if (zi != null) { zone = zi; }
10 } catch (Exception e) { /* [...] */ }
```

(a) Code vulnérable de la classe *Calendar*

```
1  // If there's a ZoneInfo object, use it for zone.
2  ZoneInfo zi = null;
3  try {
4    zi = AccessController.doPrivileged(
5    new PrivilegedExceptionAction<ZoneInfo>() {
6      public ZoneInfo run() throws Exception {
7        return (ZoneInfo) input.readObject();
8      }
9    }, CalendarAccessControlContext.INSTANCE);
10 } catch (PrivilegedActionException pae) { /* [...] */ }
11 if (zi != null) { zone = zi; }
```

(b) Code corrigé de la classe *Calendar*

FIGURE 4.9 – Comparaison entre la CVE-2008-5353 et son correctif

Cette vulnérabilité se situe dans la classe *java.util.Calendar* et plus précisément dans la méthode *readObject* en charge de désérialiser un objet *ZoneInfo* à partir d'un flot de données brutes. Pour réaliser cette opération, l'objet qui appelle cette méthode doit accéder au package *sun.util.calendar* qui est un package système de l'API Java dont l'accès nécessite le privilège *accessClassInPackage.sun.util.calendar*. En temps normal, une applet Java non signée ne dispose pas d'un tel privilège et une exception de type *SecurityException* devrait être levée d'où la nécessité d'exécuter cette portion de code avec une élévation de privilèges (cf. listing 4.9a).

En pratique, la méthode *readObject* n'a aucune connaissance sur l'objet à désérialiser. C'est pourquoi, elle se résume à allouer un objet vide en mémoire puis à y copier le flot de données bit par bit et de retourner cet objet sans en appeler le constructeur. Or le bloc *doPrivileged* implique que ces objets sont sérialisés dans un contexte privilégié et plus particulièrement avec les privilèges de la classe *Calendar* car *doPrivileged* désactive l'inspection de pile. Le but de l'exploit sera donc de désérialiser un objet malicieux pour que ce dernier échappe

8. Il s'agit du nom donné aux spécialistes des tests d'intrusion.
9. `http://newsroom.kaspersky.eu/fr/news/detail/article/statistiques-mensuelles-sur-les-programmes-malveillants-avril-2012/`

au contrôle de JAAS :

$$\text{"}Applet\text{"} \xrightarrow{invoke} \overbrace{\text{"}Calendar\text{"} \xrightarrow{invoke} \text{"}Objet\ privilégié\text{"}}^{Limite\ de\ l'inspection}$$

Nous pourrions continuer l'analyse de cette vulnérabilité mais ce qui nous intéresse se situe justement ici. La suite de l'exploit consiste principalement à utiliser une particularité d'implémentation de la méthode *readObject* pour désérialiser un chargeur de classes afin d'injecter des classes arbitraires dans le noyau de la JVM. Pour comprendre où se situe le défaut de contrôle, il faut comparer le bloc *doPrivileged* du code vulnérable avec celui du code corrigé (cf. listing 4.9b).

Dans la version vulnérable, *doPrivileged* est appelée sans privilèges particuliers ce qui, selon la documentation [10], signifie que JAAS utilisera les privilèges de la classe appelante immédiate. Sachant que *Calendar* est une classe système, le code à l'intérieur de *doPrivileged* s'exécute avec des privilèges systèmes. Il s'agit donc clairement d'une élévation de privilèges.

Le correctif, consiste à passer le privilège *accessClassInPackage.sun.util.calendar* en paramètre de *doPrivileged* (cf. ligne 9 du correctif). Selon la documentation officielle, JAAS utilisera ce privilège plutôt que ceux de la classe système *Calendar*. Il s'agit donc d'une réduction de privilèges.

Ainsi le défaut de contrôle d'accès apparaît dans le comportement de JAAS. C'est à dire que lorsque le développeur oublie de préciser les privilèges à utiliser, JAAS utilisera automatiquement le maximum de privilèges possibles ce qui est en total contradiction avec le principe de moindre privilège. Le correctif apporté résout cette vulnérabilité mais uniquement pour cet appel car le problème de fond est toujours présent. La question est donc de savoir si nous sommes en mesure d'adresser ce problème de fond.

Pour cela, suivons un raisonnement empirique. À partir de la documentation, nous savons que la classe *Calendar* implémente l'interface *Serializable* d'où l'existence de la méthode *readObject*. Toujours selon la documentation [11], cette interface indique que les objets de type *Calendar* peuvent être sérialisés et désérialisés. Donc la méthode *readObject* a pour but de désérialiser des objets de type *Calendar* et uniquement de ce type là. En d'autres termes, l'objectif de sécurité exprimé par le bloc *doPrivileged* incriminé dans la CVE-2008-5353 est de ne permettre à un objet de type Calendar de n'accéder qu'aux seuls objets du même type que lui ; d'où le privilège *accessClassInPackage.sun.util.calendar* imposé par le correctif de sécurité. En résumé, il s'agit donc de restreindre l'accès des objets *Calendar* qu'aux seuls objets dont la classe est définie dans le package *sun.util.calendar*.

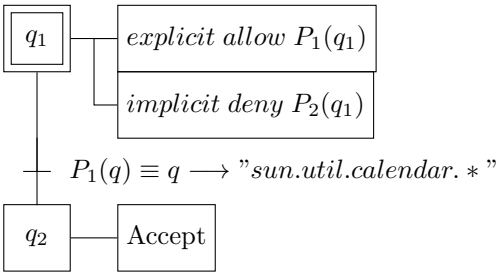
10. `http://docs.oracle.com/javase/7/docs/api/java/security/AccessController.html`
11. `http://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html`

Lorsque l'on inspecte le package Java *sun.util.calendar*, nous pouvons compter treize classes différentes ; soit treize règles de contrôle à définir. Pour simplifier les explications, nous allons plutôt considérer une expression régulière pour identifier toutes les classes de package ce qui permet de ne définir qu'une seule et unique règle :

$$allow\ "Calendar" \longrightarrow "sun.util.calendar.*"$$

Nous décomposons ensuite cette règle en privilèges élémentaires de manière à construire l'automate correspondant. Nous ajoutons à cette décomposition un privilège supplémentaire qui correspond à l'accès à une classe privilégiée de la JVM, de type *Classloader* par exemple. Nous obtenons ainsi tous les éléments nécessaires pour construire un automate :

$$\begin{aligned}
Q \equiv & \{q_1 = "Calendar",\ q_2 = "sun.util.calendar.*",\ q_3 = "Classloader"\} \\
P \equiv & \{ \\
& P_1(q_1) = q_1 \longrightarrow q_2 = q \longrightarrow "sun.util.calendar.*" \\
& P_2(q_1) = q_1 \longrightarrow q_3 = q \longrightarrow "Classloader" \\
& \}
\end{aligned}$$




Cet automate autorise donc les objets de type *Calendar* à accéder aux objets dont la classe appartient au package Java *sun.util.calendar*. En pratique donc, si le type de l'objet désérialisé par la méthode *readObject* appartient à ce package, l'objet *Calendar* pourra y accéder. Par contre, s'il s'agit d'un objet privilégié de la JVM comme un *Classloader* alors ce même automate interdira tout accès à cet objet grâce à la règle implicite de type *deny* définie à l'étape initiale $q_1$ :

$$implicit\ deny\ P_2(q_1) \equiv implict\ deny\ "Calendar" \longrightarrow "Classloader"$$

Notre approche par automate permet effectivement de réaliser une réduction de privilèges pour peu que l'on connaisse à l'avance les objets privilégiés du système

ainsi que les privilèges qui permettent d'y accéder (ici le privilège $P_2(q)$ que nous avons supposé). Aspect intéressant, la vulnérabilité que nous venons de traiter, met en lumière le privilège JAAS *(RuntimePermission, accessClassInPackage.*)* qui permet de contrôler l'accès à des objets privilégiés de la JVM. Nous pourrions donc réutiliser cette permission afin de créer des transitions dans nos automates (cf. tableau 4.5).

| Classe | Méthode |
|---|---|
| java.lang.Class | getClasses()java.lang.Class[] |
| java.lang.Class | getFields()java.lang.reflect.Field[] |
| java.lang.Class | getMethods()java.lang.reflect.Method[] |
| java.lang.Class | getConstructors()java.lang.reflect.Constructor[] |
| java.lang.Class | getField(java.lang.String)java.lang.reflect.Field |
| java.lang.Class | getMethod(*)java.lang.reflect.Method |
| java.lang.Class | getConstructor(*)java.lang.reflect.Constructor |

(a) Méthodes protégées selon [20]

$$P_8(q) = q \xrightarrow{\text{"*" } invoke \text{ "}getClasses()java.lang.Class[]\text{"}} \text{"}java.lang.Class\text{"},$$

$$P_9(q) = q \xrightarrow{\text{"*" } invoke \text{ "}getFields()java.lang.reflect.Field[]\text{"}} \text{"}java.lang.Class\text{"},$$

$$P_{10}(q) = q \xrightarrow{\text{"*" } invoke \text{ "}getMethods()java.lang.reflect.Method[]\text{"}} \text{"}java.lang.Class\text{"},$$

$$P_{11}(q) = q \xrightarrow{\text{"*" } invoke \text{ "}getConstructors()java.lang.reflect.Constructor[]\text{"}} \text{"}java.lang.Class\text{"},$$

$$P_{12}(q) = q \xrightarrow{\text{"*" } invoke \text{ "}getField(java.lang.String)java.lang.reflect.Field\text{"}} \text{"}java.lang.Class\text{"},$$

$$P_{13}(q) = q \xrightarrow{\text{"*" } invoke \text{ "}getMethod(*)java.lang.reflect.Method\text{"}} \text{"}java.lang.Class\text{"},$$

$$P_{14}(q) = q \xrightarrow{\text{"*" } invoke \text{ "}getConstructor(*)java.lang.reflect.Constructor\text{"}} \text{"}java.lang.Class\text{"}$$

(b) Transposition de la capacité en relations élémentaires

TABLE 4.5 – Réécriture de la capacité JAAS *(RuntimePermission, accessClassInPackage.*)*

### 4.2.4 Abus de l'inspection de pile

L'inspection de pile est une méthode qui consiste à analyser l'historique des appels de méthodes d'un fil d'exécution afin de déterminer si tous les objets précédemment appelés possèdent le privilège d'accéder à l'objet suivant. Comme expliqué dans l'état de l'art, l'implémentation Java de ce mécanisme est efficace tant qu'il n'existe pas d'objets privilégiés intermédiaires. En effet, la présence d'un tel objet dans la pile d'appel implique une élévation de privilèges qui a pour conséquence de désactiver l'analyse de JAAS au-delà de cet objet privilégié. L'attaque par abus de l'inspection de pile consiste justement à exploiter cette faiblesse.

**Cas général**

La section 4.1 traite déjà du cas général. Nous y avions étudié l'exemple du maçon, de la petite fille et du réfrigérateur [83] pour expliquer l'inspection de pile telle qu'implémentée par JAAS. Nous avons ainsi attesté que JAAS fonctionnait correctement lorsque la pile d'exécution contenait au plus une seule élévation de privilèges. Par contre nous avons également mis en lumière que JAAS était totalement inefficace lorsque la pile contenait au moins deux élévations de privilèges.

Le problème provient du fait que l'élévation de privilèges est une notion complexe à garantir qui nécessite une logique de contrôle plus avancée que ce que réalise JAAS en l'état. En section 3.3.4, nous avons eu l'intuition d'utiliser des automates pour justement mettre en œuvre une logique avancée qui s'est montrée très efficace pour adresser un cas d'élévation de privilèges que JAAS est incapable de traiter. Par contre il ne s'agissait là que d'une construction de pile particulière parmi toutes celles qui doivent être traitées.

C'est pourquoi, nous allons essayer de généraliser ce résultat. Pour cela, nous remarquerons qu'une pile d'exécution correspond à un chemin particulier dans un graphe d'appels. Donc, en montrant que notre approche par automate est applicable à un graphe d'appels général, nous montrerons que nous savons traiter toutes les constructions possibles de pile d'exécution et, en conséquence, les attaques sur l'inspection de pile de JAAS ne peuvent qu'échouer.

Ainsi, supposons un objet privilégié $o_P$, un objet non-privilégié $o_{NP}$, un objet qui réalise une élévation de privilèges $o_{JAAS}$ et enfin un objet quelconque du système $o_I$ dont le but est d'induire du bruit. La figure 4.10 est un graphe qui représente tous les appels possibles entre ces quatre objets. Sachant que ce graphe est totalement cyclique, il existe donc une infinité de piles d'exécution que l'on peut construire à partir de ces quatre objets. L'objectif sera de montrer que quelle que soit la pile d'exécution considérée, l'objet $o_{NP}$ ne peut avoir accès à l'objet $o_P$ sans que l'élévation de privilèges correspondante soit exprimée dans la politique.

Pour cela, nous allons utiliser un outil mathématique qui s'appelle la *fermeture transitive*. Il s'agit d'un calcul qui, dans un graphe, détermine s'il existe au moins un chemin reliant un sommet à un autre. L'avantage de ce calcul est que nous n'avons pas besoin de connaître tous les chemins possibles entre deux sommets pour savoir si il existe une connexion logique entre les deux. Dit autrement, la fermeture transitive abstrait tous les chemins entre deux sommets même si on ne les connaît pas.

Rapportée à notre cas, il est trivial de déterminer dans notre graphe d'appels qu'il existe une fermeture transitive entre les objets $o_{NP}$ et $o_P$. C'est à dire, que

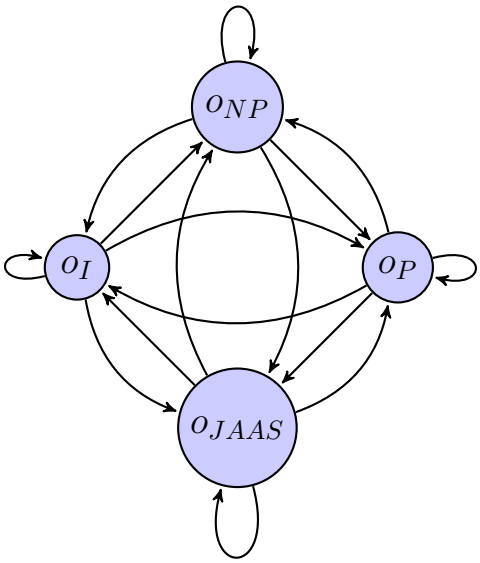



FIGURE 4.10 – Graphe des appels possibles entre quatre objets quelconques

dans le graphe de la figure 4.10 :

$$R^{trans}(o_{NP}, o_P) \neq \emptyset$$

Or, nous sommes en mesure de qualifier cette fermeture transitive. En effet, le graphe d'appels considéré ne contient que des relations inter-objets de type appel de méthodes. Donc nous pouvons affirmer que l'objet $o_{NP}$ appelle l'objet $o_P$ quelle que soit la pile d'exécution ce qui nous permet d'abstraire $R^{trans}(o_{NP}, o_P)$ par :

$$R^{trans}(o_{NP}, o_P) \equiv o_{NP} \longrightarrow o_P$$

Ce résultat est important car il signifie que l'accès d'un objet non-privilégié sur un objet privilégié est non seulement observable mais qu'en plus nous savons le formaliser. La question est maintenant de savoir si, pour une politique de sécurité donnée, il existe un automate capable de reconnaître cette relation qui est valide pour toutes les piles d'exécutions où $o_{NP}$ appelle $o_P$. Or, un automate se construit comme un graphe où les étapes sont des sommets. Ce qui signifie que le calcul de fermeture transitive entre deux étapes est faisable. Sachant que l'on observe une fermeture transitive $R^{trans}(o_{NP}, o_P)$ dans le système, notre but sera donc de trouver un automate pour lequel il existe une fermeture transitive telle que :

$$R^{trans}(Signature(o_{NP}), Signature(o_P)) \neq \emptyset$$

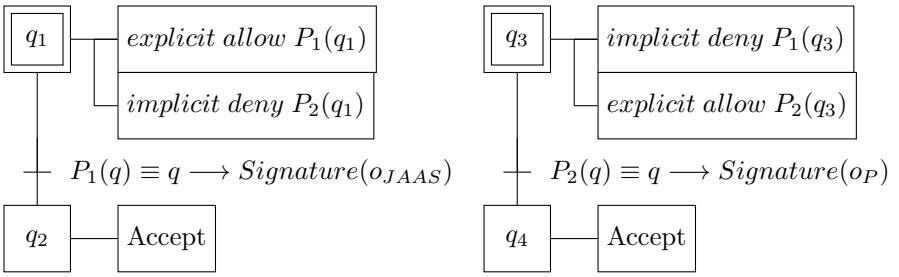
Si on identifie un tel automate alors on applique la décision de cet automate. Sinon, la relation sera automatiquement rejetée en conformité avec le principe de

*tout ce qui n'est pas explicitement autorisé est implicitement interdit.* A contrario, si l'on souhaite autoriser une élévation de privilèges entre les objets $o_{NP}$ et $o_P$ il est obligatoire d'exprimer de façon explicite la fermeture transitive correspondante, c'est à dire

$$R^{trans}(Signature(o_{NP}), Signature(o_P)) \equiv o_{NP} \longrightarrow o_P$$

Par exemple, pour les deux automates suivants il n'existe pas de fermeture transitive entre les objets $o_{NP}$ et $o_P$ (états $q_1$ et $q_4$ respectivement) donc l'élévation de privilèges n'est pas permise (implicitement interdite par le privilège $P_2(q)$). Par contre dans l'exemple du maçon, de la petite fille et du réfrigérateur une telle fermeture existe c'est pourquoi l'élévation est possible.

$$\begin{aligned} Q \equiv &\{q_1 = Signature(o_{NP}), \quad q_2 = Signature(o_I), \\ &\quad q_3 = Signature(o_{JAAS}), \quad q_4 = Signature(o_P)\} \\ P \equiv &\{P_1(q_1) = q_1 \longrightarrow Signature(o_{JAAS}), \quad P_2(q_3) = q_3 \longrightarrow Signature(o_P)\} \end{aligned}$$




*allow* $Signature(o_{NP}) \longrightarrow Signature(o_{JAAS})$ *allow* $Signature(o_{JAAS}) \longrightarrow Signature(o_P)$

$$R^{trans}(Signature(o_{NP}), Signature(o_P)) \equiv \emptyset$$

Au final, nous venons donc de montrer que peut importe la pile d'exécution considérée et quel que soit le nombre d'objets privilégiés intermédiaires, tant qu'il n'existe pas de règle explicite dans la politique de sécurité qui autorise une élévation de privilèges alors cette dernière ne sera pas possible. Cela signifie que nous devons distinguer deux cas :

— Soit l'objet intermédiaire réalise une élévation de privilèges (c-à-d $o_I = o_{JAAS}$) ce qui crée une fermeture transitive entre les deux automates ; autorisant de fait les relations de la forme $o_{NP} \longrightarrow o_{JAAS} \longrightarrow o_P$.

— Soit l'objet intermédiaire est tout à fait quelconque (c-à-d $o_I \neq o_{JAAS}$) et il n'y a pas de fermeture transitive entre les deux automates ; interdisant de fait les relations de la forme $o_{NP} \longrightarrow o_{JAAS} \longrightarrow o_P$.

Ainsi, cette notion de fermeture transitive permet de vérifier si la politique de sécurité autorise les élévations de privilèges.

**Suggestion 4.5 (Vérification de politiques)** *Soient deux objets $o_{NP}$ et $o_P$ respectivement non-privilégié et privilégié. Le principe de moindre privilèges est respecté si quelle que soit la règle de la politique :*

$$R^{trans}(Signature(o_{NP}), Signature(o_P)) \equiv \emptyset$$

**Cas Java**

Le principe général de l'inspection de pile est d'analyser les privilèges des objets appelants présents dans la pile d'exécution. Mais comme stipulée dans la documentation Java officielle [12], certaines API système utilisent une méthode de vérification différente où seule l'identité du chargeur de classe de l'appelant immédiat sera testée.

L'exploitation de la vulnérabilité CVE-2012-4681 s'appuie justement sur cette particularité pour contourner JAAS. En effet, les objets de type *Statement* permettent d'exécuter du code Java de façon interprétée ce qui implique de devoir résoudre les noms des classes et méthodes Java passés en argument. Comme expliqué dans [89], cette résolution passe par la méthode *forName* définie dans *Class*. Mais la vérification de sécurité à l'intérieur de cette méthode ne vérifie que le classloader de l'appelant immédiat et non les privilèges des objets de la pile d'exécution [12]. Sachant que l'appelant immédiat est de type *ClassFinder* défini dans le noyau de la JVM sous le package *com.sun.beans.finder*, les objets de type *Statement* peuvent donc résoudre n'importe quelle classe de la JVM y compris des classes noyaux qui peuvent ainsi être appelées par n'importe quel objet même non privilégié.

La suite de l'attaque consiste alors à accéder aux API de réflexion de bas niveau pour réussir un appel privilégié à la méthode *setSecurityManager* de la classe *System* [12]. Comme expliqué dans l'état de l'art, la classe *SecurityManager* sert d'embrayage : lorsqu'il est présent, JAAS est actif. Ainsi la méthode *setSecurityManager* permet non seulement d'activer JAAS mais aussi de le désactiver.

Plutôt que d'empêcher les appels au noyau de la JVM, qui sont obligatoires pour des raisons fonctionnelles, notre objectif sera donc d'empêcher JAAS d'être

12. `http://docs.oracle.com/javase/7/docs/api/java/lang/System.html`

désactivé de façon indue. Pour cela, il est nécessaire de contrôler les appels à la méthode *setSecurityManager* ce qui correspond à garantir la règle suivante :

$$deny * \xrightarrow{* \; invoke \; "setSecurityManager"} "java.lang.System"$$

Or, dans la section relative à l'attaque par corruption de mémoire, nous avions dû garantir une règle similaire pour contrôler l'accès à la méthode *createScreenCapture* de la classe *Robot*. Nous avions statué que la transcription de cette règle en un automate était faisable et par conséquent nous savions garantir l'application de ce type de règle. En conséquence, nous pouvons utiliser la même approche pour empêcher JAAS d'être désactivé.

Néanmoins nous devons noter que pour appeler la méthode *setSecurityManager*, le code appelant doit nécessairement disposer du privilège JAAS éponyme. De fait, nous garantissons ici encore l'application d'une règle alors même qu'il existe un privilège correspondant : la capacité JAAS *(RuntimePermission, setSecurityManager)*. À l'instar des trois cas précédents nous pouvons donc réutiliser cette permission et créer des transitions dans nos automates (cf. tableau 4.7).

| Classe | Méthode |
|---|---|
| java.lang.System | setSecurityManager(java.lang.SecurityManager)V |

(a) Méthodes protégées selon [20]

$$P_{15}(q) = q \xrightarrow{"*" \; invoke \; "setSecurityManager(java.lang.SecurityManager)V"} "java.lang.System"$$

(b) Transposition de la capacité en relations élémentaires

TABLE 4.7 – Réécriture de la capacité JAAS *(RuntimePermission, setSecurityManager)*

### 4.2.5 Discussions

Au final, les attaques par corruption de mémoire ainsi que celles par confusion de types ont lieu à bas niveau. De fait, ces deux familles d'attaques sont donc plus du ressort du système d'exploitation et du matériel que de celui du système à objets lui même. Par contre, à partir du moment où l'on contrôle les relations entre objets, nous pouvons observer les conséquences d'une attaque réussie ; c'est à dire une élévation de privilèges suivie de la prise de contrôle du système par l'attaquant. C'est donc sur ces conséquences que nous proposons d'agir en obligeant les objets du système à ne réaliser que les seules tâches qu'ils sont en droit d'accomplir. Un exemple un peu extrême : si notre politique de sécurité

autorise l'objet attaqué à accéder à un ensemble précis d'objets alors il ne pourra rien faire d'autre.

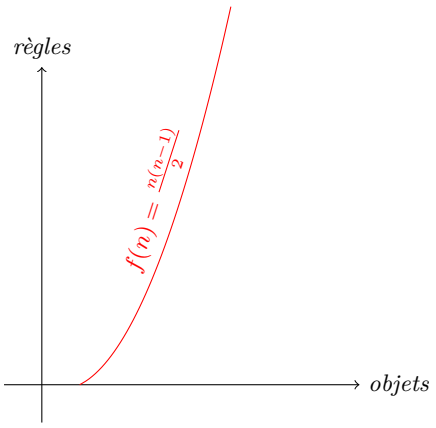
Ces deux premières catégories d'attaques sont relativement rares aujourd'hui ce qui n'est pas le cas des deux dernières. En effet, sachant qu'une partie des politiques JAAS est transcrite en code Java dans le programme, le défaut de contrôle d'accès provient principalement d'une erreur d'implémentation des objectifs de sécurité de l'application. Il ne nous est pas possible de corriger du code par une politique mais nous savons exprimer les mêmes objectifs de sécurité dans notre langage. Par ailleurs, ces défauts dans la carapace de JAAS s'accompagnent généralement d'un abus de l'inspection de pile afin de prendre le contrôle du système à objets. Or nous savons que ce type d'attaque se traduit par une élévation de privilèges observable par nos automates. Cela signifie que dans le cas où JAAS serait mis en défaut, nous savons exprimer une politique similaire à celle de JAAS et par laquelle nous pouvons autoriser ou non une élévation de privilèges précise. En d'autres termes, nos automates servent de gardes fous et respectent les objectifs de JAAS.

Nous avons essayé de mettre en lumière ces réflexions en nous appuyant sur des malwares Java. Nous avons également montré que le fait de définir des privilèges JAAS au moyen de capacités est efficace mais son implémentation n'est pas satisfaisante. En effet, chaque malware Java que nous avons étudié avait pour but de violer une règle JAAS afin d'obtenir plus de privilèges. Cela signifie que la réflexion qu'il y a eu sur la sécurité et l'accès aux API Java a déjà été faite par Sun/Oracle et semble pertinente par rapport aux organes "sensibles" d'une JVM comme par exemple le chargeur de classes. C'est cela qui nous a amenée à nous poser la question de réutiliser les politiques Java existantes comme point d'entrée de nos propres politiques. Nous présentons donc dans la section suivante, une méthode pour réutiliser les privilèges JAAS afin de les retranscrire dans une politique de contrôle applicable par nos automates.

## 4.3 Calcul automatisé de politiques de sécurité

Lorsque l'on contrôle les relations entre objets du système nous constatons assez vite qu'il est certes possible d'appliquer des règles de contrôle très fines mais surtout celles-ci peuvent être très nombreuses. En effet, si l'on suppose que tous les objets du système sont en relation deux à deux, on arrive rapidement à une explosion du nombre de règles [13] comme l'atteste la figure 4.12. Or, un système à objets, et plus particulièrement Java, peut être constitué de plusieurs dizaines de milliers d'objets en pratique. Dès lors, il n'est pas envisageable d'exprimer manuellement les règles de contrôle et c'est pour cette raison qu'une méthode automatisée de calcul de politique est nécessaire.

13. Nombre d'arrêtes dans un graphe complet constitué de $n$ sommets : $\frac{n(n-1)}{2}$




| Nombre d'objets | Nombre de règles |
|---|---|
| 5 | 10 |
| 50 | 1225 |
| 500 | 124750 |
| 5000 | 12497500 |
| ⋮ | ⋮ |

FIGURE 4.12 – Taille de la politique en fonction du nombre d'objets

Dans ce but, arrêtons nous un instant sur les outils dont nous disposons. Le chapitre 3 nous fournit un langage abstrait de modélisation grâce auquel nous pouvons décrire tout système assimilable à un système à objets. Il s'agit, en quelque sorte d'un méta-modèle de système à objets. Dans ce même chapitre nous avons alors poussé l'idée de contrôler les relations entre les objets du système. Or notre logique de contrôle par automate utilise justement comme entrée ces relations pour transformer automatiquement chaque règle abstraite en règles de contrôle (cf. figure 4.14). Il nous suffit alors de proposer un langage concret de notre modèle relationnel qui servira de sortie à nos automates.

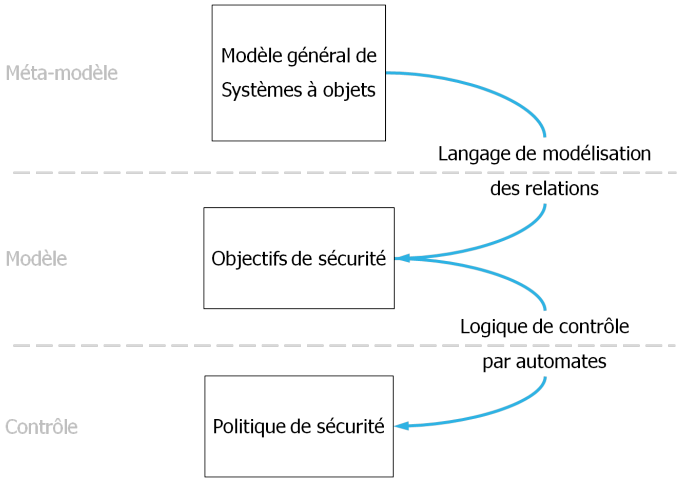



FIGURE 4.14 – Principe général de calcul de politiques.

Pour cela, nous faisons le choix de transposer les fondamentaux du *Domain & Type Enforcement* (DTE) [90] aux systèmes objets dont l'un des principaux avantages est de factoriser les règles de contrôle ce qui réduira d'autant la taille de la politique.

Ensuite, nous verrons comment extraire de JAAS les objectifs de sécurité d'une application Java afin de les transcrire automatiquement en une politique DTE.

Pour cela, nous proposons une méthode pour traduire dynamiquement le contexte de sécurité et les privilèges JAAS d'un objet Java en objectifs de sécurité au format DTE. La figure 4.15 récapitule les différentes étapes de réécriture mises en œuvre. En construisant les automates correspondants à chacun de ces objectifs nous pourrons alors calculer dynamiquement les règles de contrôle. Le chapitre suivant traite de l'implémentation pratique de cette méthode.

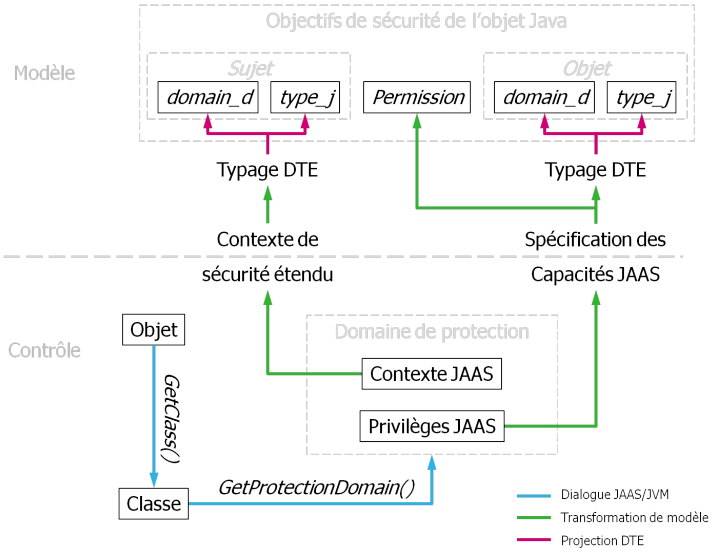



FIGURE 4.15 – Principe de transformation dynamique des domaines de protection JAAS en objectifs de sécurité au format DTE

### 4.3.1 Projection de l'approche DTE

Nous avions présenté dans l'état de l'art le concept général du *Domain & Type Enforcement* (cf. section 2.4.5). Ici, nous expliquons comment mettre en œuvre ce concept dans un système à objets. La philosophie du DTE consiste à traiter séparément l'identification des entités du système et l'expression des règles de contrôle.

#### Calcul des contextes de sécurité

Comme l'indique son nom, l'approche DTE s'appuie sur l'utilisation de types de sécurité. Fondamentalement, un type de sécurité à la même fonction que la notion de type dans un langage de programmation ; c'est à dire qu'il représente de façon abstraite la nature de l'entité à laquelle il est associé.

Or, nous avons établi dans notre modèle général sur les systèmes à objets que la signature d'un objet (cf. notation 3.10) reflète avec exactitude la nature de

cet objet quel que soit le système à objets considéré (à classes, à prototypes, ...). Ainsi, nous proposons d'étiqueter les objets du système avec un ou plusieurs types de sécurité qui sont fonction de la signature de l'objet. Cette association entre la signature d'un objet et ses types de sécurité peut, par exemple, être décrite au travers d'un fichier texte.

**Suggestion 4.6 (Étiquette et contexte DTE des objets)** *Soit $O$ un système à objets.*

1. *On définit l'ensemble $DTE$ des types de sécurité du système.*

2. *On définit l'application $Étiquette_{DTE}$ qui pour un objet $o_k \in O$ renvoie les types de sécurité $dte_i^k$ associés à la signature de l'objet $o_k \in O$ telle que :*
   $\forall o_k \in O$, $\exists$ *étiquette*$_k \subset DTE$,

$$\begin{aligned} Étiquette_{DTE} &: O \longrightarrow DTE \\ o_k &\longmapsto \quad étiquette_k \equiv \{dte_1^k, dte_2^k, \cdots, dte_d^k\} \end{aligned}$$

3. *On définit l'application $Contexte_{DTE}$ qui pour un objet $o_k \in O$ renvoie le contexte de sécurité compatible DTE de l'objet $o_k$ à partir de sa localisation telle que :*

$$\begin{aligned} Contexte_{DTE} &: L \longrightarrow DTE \\ Localisation(o_k) &\longmapsto \quad étiquette_k \end{aligned}$$

Par exemple, le tableau 4.9 associe les types primitifs Java ainsi que quelques classes de l'API avec des types de sécurité abstraits. Pour réaliser cette association, nous avons réutilisé la notation Java sur la signature des objets de la JVM [77] qui correspond au nom de l'objet lorsqu'il s'agit d'une classe ou bien à son type dans le cas contraire. Ainsi lorsque nous observerons un objet étiqueté "boolean_j" nous saurons qu'il est de valeur booléenne que ce soit le type primitif $Z$ ou bien son objet enveloppe *Boolean*. À l'inverse, ces objets devront obligatoirement être étiquetés "boolean_j". Par convention, nous proposons de suffixer les types de sécurité Java par "*_j*".

Néanmoins, cette méthode d'étiquetage ne garantit pas que l'on sera toujours en mesure de trouver au moins un type de sécurité pour chaque objet car il est tout à fait envisageable de construire un objet pour lequel il n'existe pas de règle d'étiquetage valide. Pour remédier à cette limitation, la littérature propose

| Signature Java | Type | Signature Java | Type |
|---|---|---|---|
| Ljava/lang/Object ; | object_j | Ljava/lang/Number ; | number_j |
| Ljava/lang/Class ; | class_j | Ljava/lang/String ; | string_j |
| | | | |
| Z | boolean_j | Ljava/lang/Boolean ; | boolean_j |
| B | byte_j | Ljava/lang/Byte ; | byte_j |
| C | char_j | Ljava/lang/Character ; | char_j |
| S | short_j | Ljava/lang/Short ; | short_j |
| I | int_j | Ljava/lang/Integer ; | int_j |
| J | long_j | Ljava/lang/Long ; | long_j |
| F | float_j | Ljava/lang/Float ; | float_j |
| D | double_j | Ljava/lang/Double ; | double_j |
| V | void_j | Ljava/lang/Void ; | void_j |
| E | enum_j | | |

TABLE 4.9 – Politique de labelisation minimale pour Java

d'imposer au moins un type de sécurité par défaut. Mais nous proposons une alternative pour certains systèmes à objets qui apportera une meilleure précision que cette solution "best-effort".

En effet, selon notre modélisation des systèmes à classes, la notion d'héritage est une relation de spécialisation d'une classe par une autre (cf. notation 3.24). En l'occurrence, la signature de la classe fille contient naturellement la signature de ses classes parentes. Par exemple, un *chat de gouttière* est avant tout un *chat* avant d'être *de gouttière*. Dès lors, si une classe parente transmet sa signature à ses classes filles alors les règles d'étiquetages valides pour cette classe parente s'appliquent aussi aux classes filles. Dit autrement, il existe donc une transmission naturelle des types de sécurité au travers des différentes relations d'héritage. Un raisonnement similaire peut d'ailleurs être fait concernant l'instanciation (cf. section 3.4.2) où les noms d'une classe sont naturellement transmis à ses instances. Ainsi, il suffit alors d'étiqueter la classe *Objet* pour que tous les objets du système possède *a minima* un type de sécurité au moins aussi pertinent que celui de ses classes parentes.

**Suggestion 4.7 (Transmission des types de sécurité)** *Soit $O$ un système à objets.*

1. *Les types de sécurité par défaut d'une classe $c_k \in C$ se transmettent à ses instances tels que :*
   $\forall c_k \in C$, $\acute{E}tiquette_{DTE}(Instance(c_k)) \equiv \acute{E}tiquette_{DTE}(c_k)$

2. *Les types de sécurité par défaut d'une classe $c_k \in C$ se transmettent à ses classes filles tels que :*
   $\forall c_k \in C$, $\acute{E}tiquette_{DTE}(Parent(c_k)) \subset \acute{E}tiquette_{DTE}(c_k)$

3. *Les types de sécurité par défaut d'un prototype $p_k \in O$ se transmettent à ses clones tels que :*
   $\forall p_k \in O$, $\acute{E}tiquette_{DTE}(Prototype(p_k)) \equiv \acute{E}tiquette_{DTE}(p_k)$

Ainsi, reprenons le tableau 4.9 et supposons que nous observons un objet Java de type *BigDecimal* [14] :

$$Classe(o_k) \equiv c_1 \text{ avec } Signature_{Java}(c_1) \equiv "Ljava/math/BigDecimal;"$$

Selon le premier point de la suggestion 4.7, nous estimons que l'étiquette de cet objet est équivalente à celle de sa classe :

$$o_k \in Instance(c_1) \text{ donc } \acute{E}tiquette_{DTE}(o_k) \equiv \acute{E}tiquette_{DTE}(c_1)$$

Or, nous observons que la classe *BigDecimal* hérite des classes Java *Number* et *Object* qui, selon le second point de la suggestion 4.7, lui transmettent leurs types de sécurité. Ce qui nous permet de déduire du tableau 4.9 les types de sécurité de l'objet que nous sommes en train d'observer :

$$\begin{aligned} & c_1 \rightarrowtriangle c_2 \rightarrowtriangle c_3 \\ Signature_{Java}(c_1) \equiv & "Ljava/math/BigDecimal;" \\ Signature_{Java}(c_2) \equiv & "Ljava/lang/Number;" \\ Signature_{Java}(c_3) \equiv & "Ljava/lang/Object;" \end{aligned}$$

$$\begin{aligned} \text{Donc } \acute{E}tiquette_{DTE}(c_1) \equiv & \acute{E}tiquette_{DTE}(c_1 \rightarrowtriangle c_2 \rightarrowtriangle c_3) \\ \equiv & \{"number\_j", "object\_j"\} \end{aligned}$$

$$\text{Donc } \acute{E}tiquette_{DTE}(o_k) \equiv \{"number\_j", "object\_j"\}$$

Ainsi, l'objet que nous considérons dans cet exemple possède deux types de sécurité - *number_j* et *object_j* - qui indiquent qu'il s'agit d'un objet dont la valeur est un nombre. Ce résultat est cohérent avec la documentation Java sur la classe *BigDecimal* alors même que le tableau 4.9 ne donne pas de type de sécurité explicite pour cet objet. Cette étiquette renvoie ainsi une information plus précise qu'un type de sécurité par défaut.

14. `http://docs.oracle.com/javase/7/docs/api/java/math/BigDecimal.html`

**Conséquences de la transmission des types de sécurité**

Néanmoins, notre suggestion 4.7 soulève la question de la légitimité d'un type de sécurité implicite transmis par un objet du système à un autre.

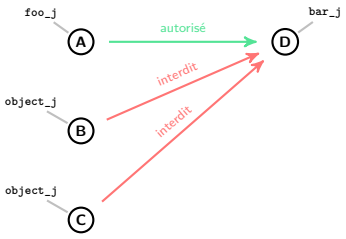
Prenons pour exemple un système constitué de quatre objets de signatures respectives $A$, $B$, $C$, et $D$ tels que :

$$O \equiv \{o_a, o_b, o_c, o_d\}$$
$$Signature(o_a) \equiv "A" \quad Signature(o_b) \equiv "B"$$
$$Signature(o_c) \equiv "C" \quad Signature(o_d) \equiv "D"$$

L'objectif sera de contrôler l'accès aux objets de signature $D$, c'est à dire $o_d$ dans notre exemple. Pour cela, nous utiliserons le tableau 4.10 pour rédiger au format texte des règles de contrôle DTE pour le système que nous considérons. L'idée est de fournir une approche pragmatique de l'écriture d'une politique DTE.

```
1 # Labelling rules
2 # (default is "object_j")
3 A      foo_j
4 D      bar_j
5
6 # Control rules
7 allow foo_j ----> bar_j
```

(a) Fichier de politique DTE




(b) Conséquence sur le système

$$Étiquette_{DTE}(o_a) \equiv "foo\_j" \qquad Étiquette_{DTE}(o_d) \equiv "bar\_j"$$
$$Étiquette_{DTE}(o_b) \equiv "object\_j" \qquad Étiquette_{DTE}(o_c) \equiv "object\_j"$$

explicit allow $o_a \longrightarrow o_d$ (car $\exists$ allow "foo_j" $\longrightarrow$ "bar_j")
implicit deny $o_b \longrightarrow o_d$ (car $\nexists$ allow "object_j" $\longrightarrow$ "bar_j")
implicit deny $o_c \longrightarrow o_d$ (car $\nexists$ allow "object_j" $\longrightarrow$ "bar_j")

(c) Politique de contrôle résultante

FIGURE 4.16 – Scénario n°1 - Privilèges d'un objet sans lignée

En l'état, Le fichier de politique présenté en figure 4.16.a autorise les objets typés *foo_j* à accéder aux objets de types *bar_j*. En l'occurrence, elle autorise explicitement les objets de signature $A$ à accéder aux objets de signature $D$ qui permet donc à l'objet $o_a$ d'accéder à l'objet $o_d$.

Mais si l'on suppose une relation de lignage/héritage entre les objets $o_a$, $o_b$ et $o_c$ alors la conséquence sur le système n'est pas la même. En effet, lorsque l'on

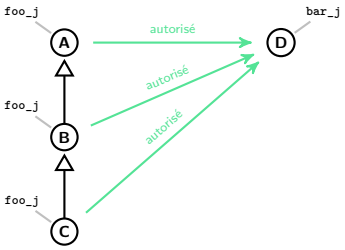
| Relation à contrôler | Syntaxe de la règle de contrôle |
|---|---|
| Interaction : | |
| $o_k \longrightarrow o_r$ | $étiquette_k - - - - > étiquette_r$ |
| $o_k \xrightarrow{invoke} o_r$ | $étiquette_k - -\{invoke\} - - > étiquette_r$ |
| Flux d'informations et de données/valeurs : | |
| $o_k \Longrightarrow o_r$ | $étiquette_k ====> étiquette_r$ |
| $o_k \xRightarrow{data} o_r$ | $étiquette_k == \{data\} ==> étiquette_r$ |
| Flux de contrôle et référence : | |
| $o_k \triangleright\triangleright o_r$ | $étiquette_k >>> étiquette_r$ |
| $o_k \dashrightarrow o_r$ | $étiquette_k ...> étiquette_r$ |

TABLE 4.10 – Proposition de syntaxe pour règles de contrôle DTE

regarde de plus près la règle *allow* définie dans le fichier de politique, il s'avère que celle-ci est explicite pour un objet d'une signature précise (cf. seconde ligne du fichier de politique) mais forcément implicite pour les objets de sa lignée ou de ses classes dérivées car la propagation des types de sécurité est implicite.

```
1 # Labelling rules
2 # (default is "object_j")
3 A      foo_j
4 D      bar_j
5
6 # Control rules
7 allow foo_j ----> bar_j
```

(a) Fichier de politique DTE




(b) Conséquence sur le système

$o_c \twoheadrightarrow o_b \twoheadrightarrow o_a$

$Étiquette_{DTE}(o_a) \equiv$ "$foo\_j$" $\quad$ $Étiquette_{DTE}(o_d) \equiv$ "$bar\_j$"

$Étiquette_{DTE}(o_b) \equiv$ "$foo\_j$" $\quad$ $Étiquette_{DTE}(o_c) \equiv$ "$foo\_j$"

explicit allow $o_a \longrightarrow o_d$ (car $\exists$ allow "foo_j" $\longrightarrow$ "bar_j")
implicit allow $o_b \longrightarrow o_d$ (car $o_b \twoheadrightarrow o_a$)
implicit allow $o_c \longrightarrow o_d$ (car $o_c \twoheadrightarrow o_b$)

Soit, selon notre suggestion 4.1 sur l'ordre de priorité des règles :

allow $o_a \longrightarrow o_d$
allow $o_b \longrightarrow o_d$
allow $o_c \longrightarrow o_d$

(c) Politique de contrôle résultante

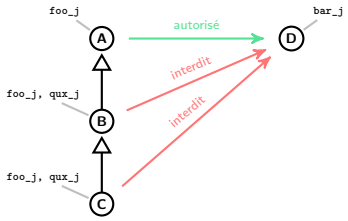
FIGURE 4.17 – Scénario n°2 - Privilèges d'un objet avec une lignée

Notre suggestion 4.7 fait que les objets $o_b$ et $o_c$ récupèrent implicitement les privilèges de l'objet $o_a$ ce qui leur permet d'accéder à l'objet $o_d$.

Nous pourrions facilement imaginer une situation où nous ne souhaitons pas que la lignée d'un objet obtienne ses privilèges. Intuitivement nous pouvons alors ajouter une règle explicite de type *deny* qui enlèverait aux objets de signature $B$ les privilèges donnés aux objets de signature $A$.

```
1 # Labelling rules
2 # (default is "object_j")
3 A      foo_j
4 D      bar_j
5 B      qux_j
6
7 # Control rules
8 allow foo_j ----> bar_j
9 deny qux_j ----> bar_j
```

(a) Fichier de politique DTE




(b) Conséquence sur le système

$o_c \twoheadrightarrow o_b \twoheadrightarrow o_a$

$\acute{E}tiquette_{DTE}(o_a) \equiv$ "$foo\_j$" $\quad \acute{E}tiquette_{DTE}(o_d) \equiv$ "$bar\_j$"

$\acute{E}tiquette_{DTE}(o_b) \equiv \{$"$foo\_j$", "$qux\_j$"$\}$ $\quad \acute{E}tiquette_{DTE}(o_c) \equiv \{$"$foo\_j$", "$qux\_j$"$\}$

explicit allow $o_a \longrightarrow o_d$ (car $\exists$ allow "foo_j" $\longrightarrow$ "bar_j")
explicit deny $o_b \longrightarrow o_d$ (car $\exists$ deny "qux_j" $\longrightarrow$ "bar_j")
implicit allow $o_b \longrightarrow o_d$ (car $o_b \twoheadrightarrow o_a$)
implicit deny $o_c \longrightarrow o_d$ (car $o_c \twoheadrightarrow o_b$)
implicit allow $o_c \longrightarrow o_d$ (car $o_c \twoheadrightarrow o_b$)

Soit, selon notre suggestion 4.1 sur l'ordre de priorité des règles :

allow $o_a \longrightarrow o_d$
deny $o_b \longrightarrow o_d$
deny $o_c \longrightarrow o_d$

(c) Politique de contrôle résultante

FIGURE 4.18 – Scénario n°3 - Non-propagation des privilèges d'un objet vers une lignée connue

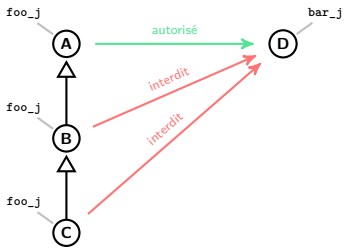
De fait, cette règle *deny* explicite désactive effectivement la propagation des privilèges donnés à l'objet $o_a$. Cette méthode suppose que l'on connaisse à l'avance la lignée de ce dernier. Cependant, la règle *deny* exprimée est valide uniquement pour cette lignée précise de l'objet $o_a$ et pas les autres. Par exemple, si on suppose un nouvel objet tel que $o_z \twoheadrightarrow o_a$ alors cet objet obtiendra automatiquement les privilèges de l'objet $o_a$ sauf à ajouter une nouvelle règle de type *deny*. Il ne s'agit donc pas d'une méthode suffisante pour maîtriser la propagation des privilèges des objets.

En complément, il faut donc pouvoir désactiver automatiquement un privilège donné à un objet sans en connaître à l'avance sa lignée. Or, si notre suggestion 4.7 induit que les règles propagées deviennent implicites, notre suggestion 4.1 pour sa part stipule qu'une règle de type *deny* est toujours prioritaire sur une règle

de type *allow*. Donc pour neutraliser une règle implicite de type *allow* il suffit de définir une règle similaire mais de type *deny*. Dit autrement, en définissant dans le fichier de politique une règle implicite de type *deny* qui se propagera en même temps que la règle explicite *allow*, on neutralise automatiquement la propagation de cette dernière.

```
1 # Labelling rules
2 # (default is "object_j")
3 A      foo_j
4 D      bar_j
5
6 # Control rules
7 explicit allow foo_j ----> bar_j
8 implicit deny foo_j ----> bar_j
```

(a) Fichier de politique DTE




(b) Conséquence sur le système

$o_c \twoheadrightarrow o_b \twoheadrightarrow o_a$

$\acute{E}tiquette_{DTE}(o_a) \equiv$ "$foo\_j$" $\quad$ $\acute{E}tiquette_{DTE}(o_d) \equiv$ "$bar\_j$"

$\acute{E}tiquette_{DTE}(o_b) \equiv$ "$foo\_j$" $\quad$ $\acute{E}tiquette_{DTE}(o_c) \equiv$ "$foo\_j$"

explicit allow $o_a \longrightarrow o_d$ (car $\exists$ explicit allow "foo_j" $\longrightarrow$ "bar_j")
implicit deny $o_a \longrightarrow o_d$ (car $\exists$ implicit deny "foo_j" $\longrightarrow$ "bar_j")
implicit allow $o_b \longrightarrow o_d$ (car $o_b \twoheadrightarrow o_a$)
implicit deny $o_b \longrightarrow o_d$ (car $o_b \twoheadrightarrow o_a$)
implicit allow $o_c \longrightarrow o_d$ (car $o_c \twoheadrightarrow o_b$)
implicit deny $o_c \longrightarrow o_d$ (car $o_c \twoheadrightarrow o_b$)

Soit, selon notre suggestion 4.1 sur l'ordre de priorité des règles :

allow $o_a \longrightarrow o_d$
deny $o_b \longrightarrow o_d$
deny $o_c \longrightarrow o_d$

(c) Politique de contrôle résultante

FIGURE 4.19 – Scénario n°4 - Non-propagation des privilèges d'un objet vers toutes ses lignées

Ainsi, nous obtenons le même résultat que précédemment à la différence près que nous n'avons pas besoin de connaître l'existence de l'objet $o_b$. Par ailleurs, on remarquera que si l'on souhaite autoriser les objets de signature $C$ à accéder aux objets de signature $D$, il suffit d'ajouter une règle *allow* explicite pour neutraliser la règle *deny* implicite. Le résultat serait alors que seuls les objets $o_a$ et $o_c$ peuvent accéder à $o_d$. De fait, même si les types de sécurité se propagent d'un objet à l'autre en suivant les lignées et les héritages, nous conservons ainsi la maîtrise sur l'attribution des privilèges.

On notera que le mot clef *final* du langage C++ (norme 2011 [91]) induit un effet analogue à la méthode que nous proposons pour conserver cette maîtrise.

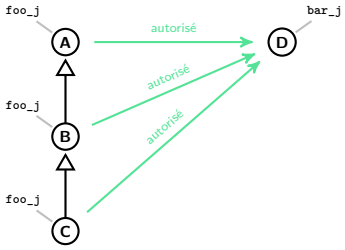
En effet, une méthode C++11 déclarée *final* ne peut être redéfinie par une classe dérivée [91]. Pour cette raison, nous proposons d'employer ce mot clef pour simplifier la syntaxe des politiques de sécurité de sorte qu'une règle DTE marquée comme finale, ne se propage pas à la lignée et classes dérivées de l'objet sur lequel elle s'applique.

```
# Labelling rules
# (default is "object_j")
A     foo_j
D     bar_j

# Control rules
allow foo_j ----> bar_j
```

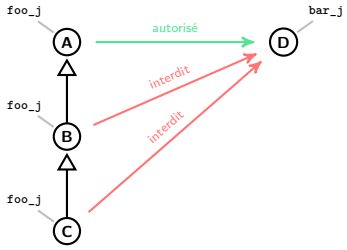
(a) Règle DTE non-finale

```
# Labelling rules
# (default is "object_j")
A     foo_j
D     bar_j

# Control rules
final allow foo_j ----> bar_j
```

(b) Règle DTE finale




(c) Conséquence de (a) sur le système




(d) Conséquence de (b) sur le système

FIGURE 4.20 – Comparaison entre une règle finale et une règle non-finale

### Expression des politiques de sécurité

Au final, le calcul des types de sécurité d'un objet est assez simple car il suffit de trouver dans la politique une association entre la signature de cet objet et un type de sécurité. Dans l'éventualité où il n'est pas possible de trouver de type de sécurité adapté pour cet objet, il suffit de lui en attribuer un par défaut comme le prévoit la littérature.

**Suggestion 4.8 (Type de sécurité par défaut)** *Le type de sécurité par défaut de tout objet d'un système à objets est "object_j".*

Dans le cas d'un système à classe, nous recommandons de parcourir l'arbre d'héritage de l'objet. Cette approche impose alors d'étiqueter obligatoirement les types primitifs ainsi que la classe *Object* pour avoir la garantie de toujours être en mesure de trouver au moins un type de sécurité pour n'importe quel objet du système. En effet, les travaux sur le parcours des graphes nous indiquent que

le parcours d'un arbre de ses feuilles jusqu'à sa racine se fait en un temps et un espace finis. Le problème de trouver un type de sécurité dans un arbre d'héritage lorsque la/les racines sont typées est donc un problème décidable pour lequel il est facile de trouver un algorithme (cf. algorithme 4).

**Entrées** : Un fichier qui associe des signatures de classe à des types de sécurité;
Une classe $c_k$ du système;
**Résultat** : L'ensemble des types de sécurité associés à la classe $c_k$ ;

**pour chaque** *classe* $c_p \in \{c_k\} \cup Parents(c_k)$ **faire**
  Calculer la signature de la classe $c_p$ telle que $signature_p = Signature(c_p)$;
  Calculer les étiquettes de sécurité de la classe $c_p$ telles que $étiquette_p = Étiquette_{DTE}(c_p)$;
  Ajouter à l'ensemble des étiquettes de sécurité de la classe $c_p$ à ceux de la classe $c_k$ tel que $étiquette_k = étiquette_k \cup étiquette_p$;
**fin**

**Algorithme 4 :** Calcul des types de sécurité d'une classe

Pour un système à prototypes il est possible d'appliquer une méthode similaire en remontant la lignée d'un objet. Il faut néanmoins prendre en compte le fait que les types de sécurité associés à un objet peuvent évoluer au gré des différentes mutations. Cette évolution devant alors être décrite dans un fichier de politique qui associerait une mutation à un type de sécurité à ajouter ou bien à enlever. En l'occurrence, la littérature indique que lorsqu'une entité change de type, elle opère une *transition* vers un nouvel ensemble de types. Cette transition étant alors conditionnée non pas à l'état de cette entité mais plutôt à ce qu'elle fait. Dans le cas d'un système à prototypes, même si la mutation est une action subie, celle-ci induit un changement suffisamment conséquent pour justifier d'une transition de types. Étant donné que nous nous sommes surtout concentrés sur les systèmes à classes, nous n'approfondirons pas ce denier aspect.

**Suggestion 4.9 (Type de sécurité par défaut)** *Chaque mutation dans un système à prototype peut induire une transition de type sur l'objet mutant.*

Par ailleurs, nous pouvons remarquer que la signature des objets est exprimée sous la forme d'une chaîne de caractères sur laquelle il serait aisée d'appliquer une expression régulière. Nous pourrions alors, avec une seule règle, typer plusieurs objets de signatures différentes comme typer tous les objets Java du paquet *java.awt.** par *graphical_element_j* par exemple.

Les règles de contrôle reflètent quant à elles les relations de notre modèle général que sont la référence, l'interaction, le flux d'activités, le flux d'information et le flux de données/valeurs. Ces relations n'étant alors plus exprimées en fonction

de la signature des objets mais plutôt en fonction de leurs types de sécurité. Ces types de sécurité étant des chaînes de caractères nous pouvons, ici aussi, mettre en œuvre des expressions régulières pour améliorer la factorisation de la politique.

**Suggestion 4.10 (Expressions régulières)** *Afin de réduire la taille des politiques de sécurité,*

1. *la signature des objets dans une règle de typage peut être une expression régulière ;*
2. *les types de sécurité dans une règle de contrôle peuvent être des expressions régulières ;*

**Suggestion 4.11 (Mots clefs des règles de contrôle)** *Les règles de contrôle peuvent être décorées à l'aide de mots clefs indiquant la nature de la règle :*

1. *allow - donne le privilège de réaliser la relation décrite par la règle ;*
2. *deny - enlève le privilège de réaliser la relation décrite par la règle ;*
3. *audit - génère une trace quand la relation décrite par la règle est réalisée ;*
4. *final - interdit la propagation du privilège décrit par la règle ;*
5. *explicit - rend explicite le privilège exprimé par la règle ;*
6. *implicit - rend implicite le privilège exprimé par la règle ;*

Ainsi, la projection des fondamentaux du *Domain & Type Enforcement* sur notre modèle de contrôle général est plutôt directe. Nous pouvons d'ailleurs apprécier le potentiel expressif des règles de contrôle au format DTE car elles tirent ici avantage de l'expressivité de notre modèle général. Néanmoins, si l'approche DTE permet effectivement de réduire le nombre de règles à définir en les factorisant, la taille des politiques reste tout de même conséquente.

### 4.3.2 Transcription de politique JAAS en politique DTE

Nous savons que notre modèle général des systèmes à classes est suffisant pour modéliser l'environnement d'exécution des machines virtuelles Java. Dès lors nous pouvons utiliser notre langage de modélisation pour exprimer les objectifs de sécurité d'une application Java pour les transformer ensuite en règles de

contrôle DTE et/ou JAAS par le biais d'automates ; cela correspond aux flèches descendantes de la figure 4.21. La difficulté sera alors de retrouver les objectifs de sécurité d'une application Java à partir de leurs implémentations, c'est à dire à partir des règles de contrôle JAAS ; la flèche montante de gauche sur la figure 4.21 symbolise cette étape. En dernière étape, nous pourrons utiliser notre logique de contrôle par automate pour transformer ces objectifs en règles de contrôle au format DTE.

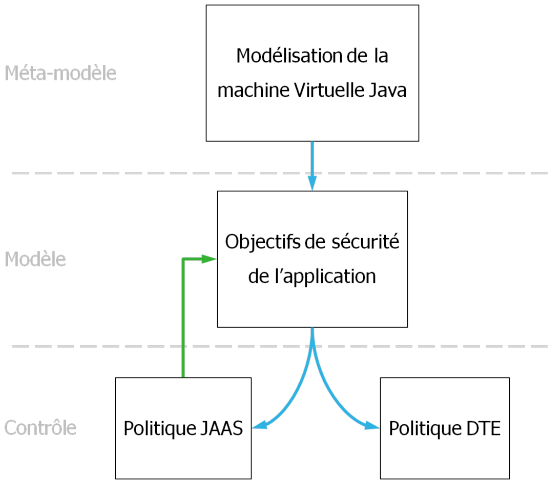



FIGURE 4.21 – Calcul automatisé de politiques appliqué à Java

L'idée est de s'appuyer sur la documentation officielle et notamment [92] qui explique qu'une politique Java consiste à associer un triplet *(codesource, codesigners, principals)* à un ensemble de permissions JAAS. Par analogie avec notre approche de contrôle, ce triplet représente la signature JAAS d'un objet de la JVM auquel la politique Java donne un ensemble de privilèges. Or, nous avions expliqué dans la section 2.3.3 de l'état de l'art que chaque permission JAAS est en réalité une capacité dont les spécifications sont définies par [20].

Ainsi, nous avons d'un côté l'expression de politiques Java sous la forme d'un ensemble $D$ de signatures JAAS et un ensemble $C$ de capacités (cf. figure 4.22). De l'autre, nous avons notre logique de contrôle par automates qui s'appuie sur un ensemble $Q$ de signatures d'objets et un ensemble $P$ de relations élémentaires. En d'autres termes, chaque règle d'une politique exprimée en langage JAAS est fonction d'une variable $d$ qui est la signature JAAS de l'entité *Sujet* alors que, dans notre langage, l'entité *Sujet* dépend d'une variable $q$ qui est la signature d'un objet. C'est pourquoi, nous devons trouver un moyen de relier une signature JAAS à une signature d'objet dans notre langage afin de pouvoir réécrire les capacités JAAS en relations inter-objets.

$$
\begin{array}{l}
D \equiv\{ \\
\quad d_1 = (codesource_1, codesigners_1, principals_1), \\
\quad d_2 = (codesource_2, codesigners_2, principals_2), \\
\quad \cdots \\
\quad d_i = (codesource_i, codesigners_i, principals_i) \\
\} \\
C \equiv\{C_1(d), C_2(d), \cdots, C_j(d)\}
\end{array}
$$

(a) Éléments utilisés par JAAS

$$
\begin{array}{l}
Q \equiv\{ \\
\quad q_1 = signature_1, \\
\quad q_2 = signature_2, \\
\quad \cdots \\
\quad q_i = signature_i \\
\} \\
P \equiv\{P_1(q), P_2(q), \cdots, P_j(q)\}
\end{array}
$$

(b) Éléments utilisés par nos automate

FIGURE 4.22 – Comparaison de langage entre JAAS et nos automates

**Signature et contexte de sécurité JAAS**

Dans la section 2.3.3 de l'état de l'art nous avons expliqué que lorsqu'une règle de la politique Java s'applique sur un objet de la JVM, JAAS crée automatiquement pour cet objet un *domaine de protection*. Il s'agit d'un objet Java particulier qui contient cinq informations au sujet de la classe de cet objet : son chargeur de classe, l'URL de son binaire, la liste des certificats qui valident son intégrité, l'ensemble des principaux propriétaires de cette classe, et enfin une liste de permissions JAAS.

Or, par définition, un chargeur de classes est un espace mémoire dédié, ce qui correspond à une localisation "gros grain" dans la JVM [15]. De plus, les trois champs qui suivent (URL/signataires/principaux) servent à identifier le sujet d'une règle Java et ont donc par définition le même rôle qu'une signature. Cela étant, puisque JAAS exprime une information de localisation et une information de signature, il est facile de déduire que les deux réunis forment un contexte de sécurité "gros grain" du fait de l'imprécision sur la localisation.

**Notation 4.1 (Signature JAAS)** *Soit O l'environnement objets de la Machine Virtuelle Java.*

1. *On définit l'ensemble CLASSPATH constitué des chemins possibles pour l'accès à une ressource de la JVM.*

2. *On définit l'ensemble SIGNERS constitué des identités possibles pour un signataire de code.*

15. La mémoire de la JVM peut être vue comme une commode apothicaire où chaque tiroir est un chargeur de classe : JAAS sait exactement dans quel tiroir se situe une classe et ses instances mais il ne sait dire où à l'intérieur de ce tiroir.

3. *On définit l'ensemble PRINCIPALS constitué des identités possibles pour une entité JAAS.*

4. *On définit l'application* $Signature_{JAAS}$ *qui renvoie la signature JAAS d'une classe* $c_k \in C$ *telle que :*

$$\begin{aligned} Signature_{JAAS} \colon C &\longrightarrow CLASSPATH \times SIGNERS \times PRINCIPALS \\ c_k &\longmapsto (codesource_k,\ codesigners_k,\ principals_k) \end{aligned}$$

*Remarque : Les valeurs possibles des éléments des ensembles CLASSPATH, SIGNERS et PRINCIPALS dépendent de la Machine Virtuelle Java [92].*

**Notation 4.2 (Contexte de sécurité JAAS)** *Soit* $O$ *l'environnement objets de la Machine Virtuelle Java.*

1. *On définit l'application* $Localisation_{JAAS}$ *qui renvoie le chargeur de classes* $classloader_k \in O$ *d'une classe* $c_k \in C$ *telle que :*

$$\begin{aligned} Localisation_{JAAS} \colon C &\longrightarrow O \\ c_k &\longmapsto Classloader(c_k) \end{aligned}$$

2. *On définit l'application* $Contexte_{JAAS}$ *qui renvoie le contexte de sécurité JAAS d'une classe* $c_k \in C$ *telle que :*

$$\begin{aligned} Contexte_{JAAS} \colon C &\longrightarrow N \times T \times CLASSPATH \times SIGNERS \times PRINCIPALS \\ c_k &\longmapsto (Signature(Classloader(c_k)),\ Signature_{JAAS}(c_k)) \end{aligned}$$

Ainsi quand on observe un objet Java nous pouvons déduire que celui-ci possède deux contextes de sécurité exprimés dans des langages différents : celui de JAAS et le nôtre. Le lien entre ces deux contextes étant alors réalisé par l'objet qui implémente le *domaine de protection* et dont la valeur est automatiquement calculée par JAAS au moment du chargement de la classe. Cela signifie que pour lier ces deux contextes nous pouvons considérer un contexte de sécurité étendu propre à Java qui intègre ces deux éléments.

**Notation 4.3 (Contexte de sécurité Java étendu)** *Soit $O$ l'environnement objets de la Machine Virtuelle Java.*

*On définit l'application $Contexte_{Java}$ qui renvoie le contexte de sécurité d'un objet $o_k \in O$ de la JVM à partir de sa localisation telle que :*

$$\begin{aligned} Contexte_{Java} : L &\longrightarrow (N \times T) \times (N \times T) \times CLASSPATH \times SIGNERS \times PRINCIPALS \\ Localisation(o_k) &\longmapsto (Contexte(Localisation(o_k)), Contexte_{JAAS}(Classe(o_k))) \end{aligned}$$

Prenons l'exemple d'une applet. La méthode standard pour interroger JAAS sur le domaine de protection d'un objet consiste à appeler la méthode *GetProtectionDomain*[16] de l'API Java. La figure 4.23 présente un programme Java autonome qui affiche son propre domaine de protection par ce biais. Les tableaux 4.11 donnent le résultat de l'exécution de ce même programme en tant qu'applet Java.

Ainsi, lorsque ce programme est embarqué dans une page HTML [93], on remarque rapidement qu'une applet Java est principalement autorisée à lire les variables d'environnement de la JVM. Les privilèges JAAS *getProtectionDomain* et *getPolicy* ont étés rajoutés par nos soins pour extraire le domaine de protection du programme ainsi que ses privilèges car les applets Java ne peuvent normalement pas réaliser ce type d'opérations privilégiées. Plus particulièrement, JAAS nous renseigne surtout sur le contexte de sécurité de la classe principale de notre programme :

$$O \equiv \{\cdots, o_1, c_1, \cdots\} \qquad Classe(o_1) \equiv c_1$$

$$\begin{aligned} Signature(o_1) &\equiv Signature_{Java}(o_1) \equiv "SEJavaProgram" \\ Contexte(Localisation(o_1)) &\equiv Signature(o_1) \end{aligned}$$

$$\begin{aligned} Signature_{JAAS}(c_1) &\equiv ("\texttt{file://C:/users/bvenelle/dev/SEJava}", \emptyset, \emptyset) \\ Contexte_{JAAS}(c_1) &\equiv (Signature(Classloader(c_1)),\ Signature_{JAAS}(c_1)) \\ &\equiv ("sun.plugin2.applet.Applet2ClassLoader", \\ &\quad "\texttt{file://C:/users/bvenelle/dev/SEJava}", \emptyset, \emptyset) \end{aligned}$$

$$\begin{aligned} Contexte_{Java}(Localisation(o_1)) &\equiv (Contexte(Localisation(o_1)),\ Contexte_{JAAS}(Classe(o_1))) \\ &\equiv (Signature(o_1),\ Contexte_{JAAS}(c_1)) \end{aligned}$$

16. `http://docs.oracle.com/javase/7/docs/api/java/lang/Class.html#getProtectionDomain()`

$$\equiv(\text{"}SEJavaProgram\text{"}, \text{"}sun.plugin2.applet.Applet2ClassLoader\text{"},$$
$$\text{"}\texttt{file://C:/users/bvenelle/dev/SEJava}\text{"}, \emptyset, \emptyset)$$

Grâce à ce contexte de sécurité étendu, nous disposons d'un moyen pour connaître avec précision quel objet de la JVM a le droit d'accéder à quel autre. En effet, le *domaine de protection* JAAS attaché aux objets de la JVM nous renvoie également une version consolidée de la politique JAAS appliquée à cet objet précis. Par exemple, le tableau 4.12b nous indique que notre applet dispose des capacités *(PropertyPermission, *, read)* et *(RuntimePermission, accessClassInPackage.*)*. À l'aide des spécifications de sécurité de ces deux capacités JAAS [20], nous pouvons déduire des tableaux 4.13 et 4.5 qu'elles sont les méthodes que notre applet est en droit d'accéder. La figure 4.24 propose par exemple une réécriture des règles JAAS de type *grant* correspondantes en règles de contrôle dans notre langage.

Pour information, nous pourrions intuitivement penser qu'un travail similaire pourrait être réalisé directement sur les fichiers de politique Java. En effet, le champ *codesource* de chaque règle fait référence à une URL pointant vers un ensemble de fichiers *.class* qu'il suffit de scanner pour déterminer les classes concernées par la règle. Malheureusement un contexte de sécurité JAAS contient des informations calculées dynamiquement telles que la liste des principaux et il en va de même pour les politiques associées à cet objet comme l'atteste l'implémentation de la méthode *addDefaultPermissions* de la classe *Plugin2ClassLoader* dont hérite *Applet2ClassLoader*. La seule option possible est donc effectivement de laisser JAAS faire "sa magie" et de s'appuyer sur le domaine de protection de l'objet comme nous l'avons fait en récupérant dynamiquement le *domaine de protection* de chaque objet à l'exécution de l'application Java.

#### Réécriture des politiques JAAS

Comme expliqué dans la section 2.3.3 de l'état de l'art relative à JAAS, une permission JAAS est l'expression d'une capacité. Les spécifications Java, et plus particulièrement [20], nous renseignent sur les méthodes de l'API Java protégées par chacune de ces capacités. Ainsi, lorsque JAAS octroie une capacité à un objet via son *domaine de protection*, nous pouvons déduire les relations élémentaires que JAAS autorise en réalité comme nous l'avons fait précédemment avec les permissions JAAS *(AWTPermission, createRobot)*, *(AWTPermission, readDisplayPixels)*, *(RuntimePermission, createClassloader)*, *(RuntimePermission, accessClassInPackage.*)*, *(RuntimePermission, setSecurityManager)* et *(PropertyPermission, *, read)* au travers des tableaux 4.1, 4.3, 4.5, 4.7 et 4.13 respectivement.

Ainsi, la réécriture d'une politique JAAS est plutôt simple à partir du moment où nous avons accès à ses spécifications. Malgré tout, certaines capacités JAAS

```java
// Import the required Java libraries
import java.lang.*;
import java.security.*;
import java.applet.Applet;
import java.awt.*;

// The program code
public class SEJavaProgram extends Applet {

  // Required variables
  protected String jaas_domain = null;

  public static void main(String[] argv) {

    // Display assigned JAAS protection domain
    SEJavaProgram program = new SEJavaProgram();
    System.out.println(program.jaas());
  }

  public void paint(Graphics g) {

    // Display assigned JAAS protection domain
    int offset = 10; for (String line : this.jaas().split("\n"))
      g.drawString(line, 10, offset += 15);
  }

  public synchronized String jaas() {

    if (null != this.jaas_domain) return this.jaas_domain;

    // Format JAAS protection domain
    this.jaas_domain = "--------- Active JAAS domain ------------\n";

    try {

      // Get reference on program's class
      Class program = this.getClass();

      // Forge JAAS protection domain representation
      ProtectionDomain domain = program.getProtectionDomain();
      this.jaas_domain += domain.toString();

      // Forge granted JAAS permissions representation
      //this.jaas_domain += "--------- Active JAAS policy ------------\n";
      //Policy policy = Policy.getPolicy();
      //PermissionCollection permissions = policy.getPermissions(domain);
      //this.jaas_domain += permissions.toString();

    } catch(java.lang.SecurityException e) {

      // Some JAAS capabilities are required
      this.jaas_domain += "Please grant the following permissions:\n"
        + "  java.lang.RuntimePermission  \"getProtectionDomain\"\n"
        + "  java.security.SecurityPermission  \"getPolicy\"\n";
    }

    // Job is done
    this.jaas_domain += "\n-----------------------------------------";
    return this.jaas_domain;
  }
}
```

FIGURE 4.23 – Programme Java qui affiche son propre domaine de protection

| Clé | Valeur |
|---|---|
| Classloader | sun.plugin2.applet.Applet2ClassLoader |
| Codesource | file ://C :/users/bvenelle/dev/SEJava |
| Codesigners | n/a |
| Principals | n/a |

(a) Contexte de sécurité JAAS

| Permission | Cible | Actions |
|---|---|---|
| SecurityPermission | getPolicy | |
| PropertyPermission | browser | read |
| PropertyPermission | browser.vendor | read |
| PropertyPermission | browser.version | read |
| PropertyPermission | file.separator | read |
| PropertyPermission | http.agent | read |
| PropertyPermission | java.class.version | read |
| PropertyPermission | java.specification.name | read |
| PropertyPermission | java.specification.vendor | read |
| PropertyPermission | java.specification.version | read |
| PropertyPermission | java.vendor | read |
| PropertyPermission | java.vendor.url | read |
| PropertyPermission | java.version | read |
| PropertyPermission | java.vm.name | read |
| PropertyPermission | java.vm.specification.name | read |
| PropertyPermission | java.vm.specification.vendor | read |
| PropertyPermission | java.vm.specification.version | read |
| PropertyPermission | java.vm.vendor | read |
| PropertyPermission | java.vm.version | read |
| PropertyPermission | javapi.* | read,write |
| PropertyPermission | javaplugin.version | read |
| PropertyPermission | javaplugin.vm.options | read |
| PropertyPermission | javaws.* | read,write |
| PropertyPermission | jnlp.* | read,write |
| PropertyPermission | line.separator | read |
| PropertyPermission | mrj.version | read |
| PropertyPermission | os.arch | read |
| PropertyPermission | os.name | read |
| PropertyPermission | os.version | read |
| PropertyPermission | path.separator | read |
| SecureCookiePermission | origin.file :// :-1 | |
| SocketPermission | localhost :1024 | listen,resolve |
| RuntimePermission | accessClassInPackage.sun.audio | |
| RuntimePermission | getProtectionDomain | |
| RuntimePermission | stopThread | |
| FilePermission | /C :/users/bvenelle/dev/SEJava/- | read |

(b) Privilèges JAAS

TABLE 4.11 – Exemple de domaine de protection par défaut des applets sous Java 1.7u15

$$
\begin{array}{l}
\{ \\
\quad grant\ "SEJavaProgram"\ jaas.capability("RuntimePermission", "accessClassInPackage.*") \\
\quad grant\ "SEJavaProgram"\ jaas.capability("PropertyPermission", "*", "read") \\
\} \equiv \{ \\
\quad allow\ P_8("SEJavaProgram"),\quad allow\ P_9("SEJavaProgram"),\quad allow\ P_{10}("SEJavaProgram"), \\
\quad allow\ P_{11}("SEJavaProgram"),\quad allow\ P_{12}("SEJavaProgram"),\quad allow\ P_{13}("SEJavaProgram"), \\
\quad allow\ P_{14}("SEJavaProgram"),\quad allow\ P_{16}("SEJavaProgram"),\quad allow\ P_{17}("SEJavaProgram"), \\
\quad allow\ P_{18}("SEJavaProgram"),\quad allow\ P_{19}("SEJavaProgram"),\quad allow\ P_{20}("SEJavaProgram"), \\
\quad allow\ P_{21}("SEJavaProgram"),\quad allow\ P_{22}("SEJavaProgram"),\quad allow\ P_{23}("SEJavaProgram"), \\
\quad allow\ P_{24}("SEJavaProgram") \\
\} \equiv \{ \\
\quad allow\ "SEJavaProgram" \xrightarrow{"*"\ invoke\ "getClasses()java.lang.Class[]"} "java.lang.Class", \\
\quad allow\ "SEJavaProgram" \xrightarrow{"*"\ invoke\ "getFields()java.lang.reflect.Field[]"} "java.lang.Class", \\
\quad allow\ "SEJavaProgram" \xrightarrow{"*"\ invoke\ "getMethods()java.lang.reflect.Method[]"} "java.lang.Class", \\
\quad allow\ "SEJavaProgram" \xrightarrow{"*"\ invoke\ "getConstructors()java.lang.reflect.Constructor[]"} "java.lang.Class", \\
\quad allow\ "SEJavaProgram" \xrightarrow{"*"\ invoke\ "getField(java.lang.String)java.lang.reflect.Field"} "java.lang.Class", \\
\quad allow\ "SEJavaProgram" \xrightarrow{"*"\ invoke\ "getMethod(*)java.lang.reflect.Method"} "java.lang.Class", \\
\quad allow\ "SEJavaProgram" \xrightarrow{"*"\ invoke\ "getConstructor(*)java.lang.reflect.Constructor"} "java.lang.Class" \\
\quad allow\ "SEJavaProgram" \xrightarrow{"*"\ invoke\ "setDesignTime(Z)V"} "Beans", \\
\quad allow\ "SEJavaProgram" \xrightarrow{"*"\ invoke\ "setGuiAvailable(Z)V"} "Beans", \\
\quad allow\ "SEJavaProgram" \xrightarrow{"*"\ invoke\ "setBeanInfoSearchPath(String[])V"} "Introspector", \\
\quad allow\ "SEJavaProgram" \xrightarrow{"*"\ invoke\ "registerEditor(Class,Class)V"} "PropertyEditorManager", \\
\quad allow\ "SEJavaProgram" \xrightarrow{"*"\ invoke\ "setEditorSearchPath(String[])V"} "PropertyEditorManager", \\
\quad allow\ "SEJavaProgram" \xrightarrow{"*"\ invoke\ "getProperties()Properties"} "System", \\
\quad allow\ "SEJavaProgram" \xrightarrow{"*"\ invoke\ "setProperties(Properties)V"} "System", \\
\quad allow\ "SEJavaProgram" \xrightarrow{"*"\ invoke\ "getProperty(String)String"} "System", \\
\quad allow\ "SEJavaProgram" \xrightarrow{"*"\ invoke\ "getProperty(String,String)String"} "System" \\
\}
\end{array}
$$

FIGURE 4.24 – Réécriture partielle de la politique JAAS du domaine de protection du tableau 4.11

| Classe | Méthode |
|---|---|
| java.beans.Beans | setDesignTime(Z)V |
| java.beans.Beans | setGuiAvailable(Z)V |
| java.beans.Introspector | setBeanInfoSearchPath(String[])V |
| java.beans.PropertyEditorManager | registerEditor(Class, Class)V |
| java.beans.PropertyEditorManager | setEditorSearchPath(String[])V |
| java.lang.System | getProperties()Properties |
| java.lang.System | setProperties(Properties)V |
| java.lang.System | getProperty(String)String |
| java.lang.System | getProperty(String, String)String |

(a) Méthodes protégés par la capacité selon [20]

$$P_{16}(q) = q \xrightarrow{\text{"*" invoke "setDesignTime(Z)V"}} \text{"java.beans.Beans"},$$
$$P_{17}(q) = q \xrightarrow{\text{"*" invoke "setGuiAvailable(Z)V"}} \text{"java.beans.Beans"},$$
$$P_{18}(q) = q \xrightarrow{\text{"*" invoke "setBeanInfoSearchPath(String[])V"}} \text{"java.beans.Introspector"},$$
$$P_{19}(q) = q \xrightarrow{\text{"*" invoke "registerEditor(Class,Class)V"}} \text{"java.beans.PropertyEditorManager"},$$
$$P_{20}(q) = q \xrightarrow{\text{"*" invoke "setEditorSearchPath(java.lang.String[])V"}} \text{"java.beans.PropertyEditorManager"},$$
$$P_{21}(q) = q \xrightarrow{\text{"*" invoke "getProperties()Properties"}} \text{"java.lang.System"},$$
$$P_{22}(q) = q \xrightarrow{\text{"*" invoke "setProperties(Properties)V"}} \text{"java.lang.System"},$$
$$P_{23}(q) = q \xrightarrow{\text{"*" invoke "getProperty(String)String"}} \text{"java.lang.System"},$$
$$P_{24}(q) = q \xrightarrow{\text{"*" invoke "getProperty(String,String)String"}} \text{"java.lang.System"}$$

(b) Capacité réécrite selon notre langage de relation

TABLE 4.13 – Réécriture de la capacité *(PropertyPermission, *, read)*

ne sont pas spécifiées par [20] comme par exemple la permission *SecureCookiePermission* donnée par JAAS à notre applet Java précédente. Dans ce cas précis il nous est donc difficile d'en proposer une réécriture. Nous n'avons pas résolu cette question mais notre principale piste de réflexion est de surveiller les permissions vérifiées par JAAS à chacun de ses contrôles. En effet, le moniteur de référence de JAAS est appelé directement depuis la méthode à protéger (cf. section 2.3.4 de l'état de l'art), il suffirait donc d'instrumenter ce composant pour approximer les appels de méthodes concernés par une capacité.

### Génération des règles de contrôle

Lorsque nous réécrivons la politique d'un *domaine de protection* JAAS, nous mettons surtout en lumière les objectifs de sécurité de ce *domaine de protection* ; c'est à dire les seules relations qu'un objet de ce *domaine de protection* est en droit de réaliser. Par exemple, lorsque l'on reprend la politique de la figure 4.24 relative à notre applet, nous pouvons construire un automate correspondant au *domaine de protection* de notre applet qui utilise soit le langage JAAS (cf. automate 4.25.a), soit le nôtre (cf. automate 4.25.b).

Or, ces automates sont pour l'instant incomplets car ils ne tiennent compte que des capacités dont dispose notre applet. C'est pourquoi nous devons ajouter à ces derniers les capacités qui ont été refusées par JAAS en suivant la même approche que dans la section 4.1.3 en début de ce chapitre. Pour plus de visibilités, nous ne considérerons que les capacités *(AWTPermission, createRobot)*, *(AWTPermission, readDisplayPixels)*, *(RuntimePermission, createClassloader)* et *(RuntimePermission, setSecurityManager)*. En effet, les spécifications Java [20] définissent plus de 70 capacités $C_i(d)$ différentes qui donnent lieu à plus de 200 permissions $P_j(q)$ après réécriture et donc des automates assez vastes.

La figure 4.26 donnent les automates de contrôle ainsi obtenus. Nous pouvons facilement constater que l'automate 4.26a utilise le mot clef *refuse* pour symboliser le fait que JAAS n'a pas donné les capacités $C_3$ à $C_6$ même si le langage JAAS ne définit pas ce mot clef. Le second constat est que l'automate 4.26b est plutôt simple car il n'est constitué que d'un état initial ($q_1$) et de plusieurs états finaux ($q_2$ à $q_6$). Cela signifie que nous pouvons directement appliquer la politique de l'état initial $q_1$ pour contrôler les relations de notre applet :

$$allow\ "SEJavaProgram" \xrightarrow{"*"\ invoke\ "getClasses()java.lang.Class[]"} "java.lang.Class",$$

$$allow\ "SEJavaProgram" \xrightarrow{"*"\ invoke\ "getFields()java.lang.reflect.Field[]"} "java.lang.Class",$$

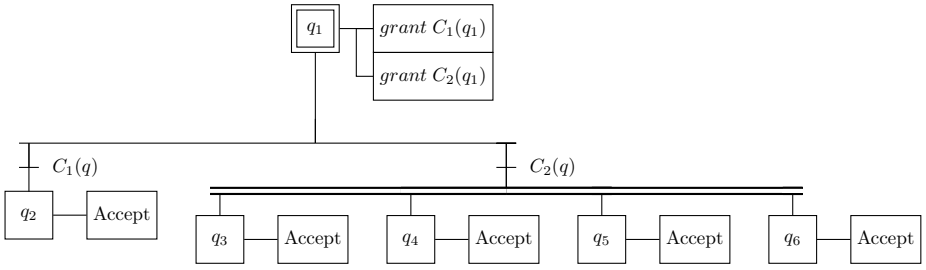
$$allow\ "SEJavaProgram" \xrightarrow{"*"\ invoke\ "getMethods()java.lang.reflect.Method[]"} "java.lang.Class",$$

$$allow\ "SEJavaProgram" \xrightarrow{"*"\ invoke\ "getConstructors()java.lang.reflect.Constructor[]"} "java.lang.Class",$$

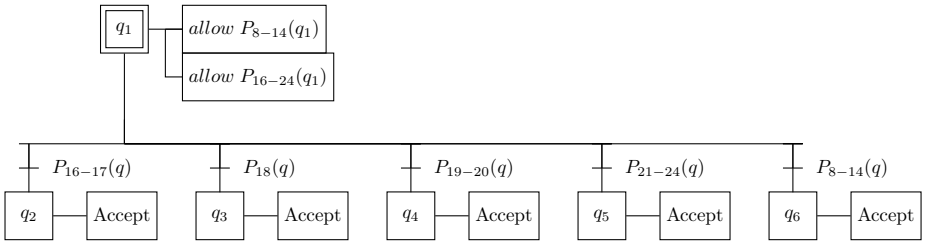
$$allow\ "SEJavaProgram" \xrightarrow{"*"\ invoke\ "getField(java.lang.String)java.lang.reflect.Field"} "java.lang.Class",$$

$$allow\ "SEJavaProgram" \xrightarrow{"*"\ invoke\ "getMethod(*)java.lang.reflect.Method"} "java.lang.Class",$$

$$allow\ "SEJavaProgram" \xrightarrow{"*"\ invoke\ "getConstructor(*)java.lang.reflect.Constructor"} "java.lang.Class",$$




(a) Automate reflétant la politique JAAS




(b) Automate reflétant les objectifs de sécurité

$$
\begin{aligned}
Q \equiv \{ & \\
& q_1 = \text{"}SEJavaProgram\text{"}, \quad q_2 = \text{"}java.lang.Class\text{"}, \\
& q_3 = \text{"}java.beans.Beans\text{"}, \quad q_4 = \text{"}java.beans.Introspector\text{"}, \\
& q_5 = \text{"}java.beans.PropertyEditorManager\text{"}, \quad q_6 = \text{"}java.lang.System\text{"} \\
\} & \\
P \equiv \{ & \\
& P_8(q_1), \quad P_9(q_1), \quad P_{10}(q_1), \quad P_{11}(q_1), \quad P_{12}(q_1), \quad P_{13}(q_1), \\
& P_{14}(q_1), \quad P_{16}(q_1), \quad P_{17}(q_1), \quad P_{18}(q_1), \quad P_{19}(q_1), \quad P_{20}(q_1), \\
& P_{21}(q_1), \quad P_{22}(q_1), \quad P_{23}(q_1), \quad P_{24}(q_1) \\
\} & \\
C \equiv \{ & \\
& C_1(q_1) = q_1 \; jaas.capability(\text{"}RuntimePermission\text{"}, \text{"}accessClassInPackage.*\text{"}), \\
& C_2(q_1) = q_1 \; jaas.capability(\text{"}PropertyPermission\text{"}, \text{"}*\text{"}, \text{"}read\text{"}) \\
\} &
\end{aligned}
$$

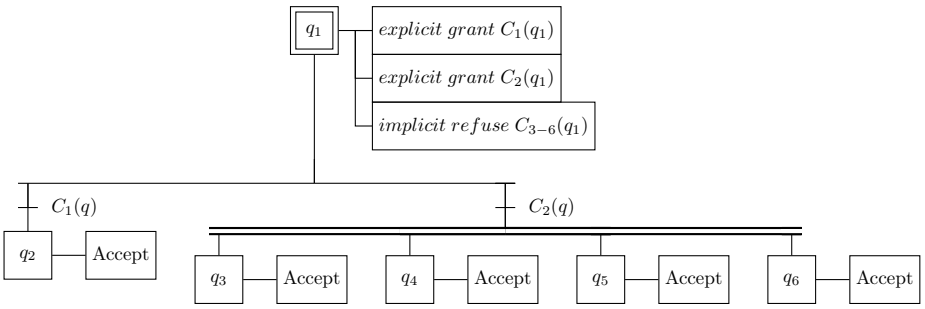
FIGURE 4.25 – Automates du *domaine de protection* relatif aux applets Java

$$allow\ "SEJavaProgram" \xrightarrow{"*"\ invoke\ "setDesignTime(Z)V"} "Beans",$$
$$allow\ "SEJavaProgram" \xrightarrow{"*"\ invoke\ "setGuiAvailable(Z)V"} "Beans",$$
$$allow\ "SEJavaProgram" \xrightarrow{"*"\ invoke\ "setBeanInfoSearchPath(String[])V"} "Introspector",$$
$$allow\ "SEJavaProgram" \xrightarrow{"*"\ invoke\ "registerEditor(Class,Class)V"} "PropertyEditorManager",$$
$$allow\ "SEJavaProgram" \xrightarrow{"*"\ invoke\ "setEditorSearchPath(String[])V"} "PropertyEditorManager",$$
$$allow\ "SEJavaProgram" \xrightarrow{"*"\ invoke\ "getProperties()Properties"} "System",$$
$$allow\ "SEJavaProgram" \xrightarrow{"*"\ invoke\ "setProperties(Properties)V"} "System",$$
$$allow\ "SEJavaProgram" \xrightarrow{"*"\ invoke\ "getProperty(String)String"} "System",$$
$$allow\ "SEJavaProgram" \xrightarrow{"*"\ invoke\ "getProperty(String,String)String"} "System",$$
$$deny\ "SEJavaProgram" \xrightarrow{"*"\ invoke\ "<init>()V"} "java.awt.Robot",$$
$$deny\ "SEJavaProgram" \xrightarrow{"*"\ invoke\ "<init>(java.awt.GraphicsDevice)V"} "java.awt.Robot",$$
$$deny\ "SEJavaProgram" \xrightarrow{"*"\ invoke\ "setComposite(java.awt.Composite)V"} "java.awt.Graphics2d",$$
$$deny\ "SEJavaProgram" \xrightarrow{"*"\ invoke\ "<init>()V"} "ClassLoader",$$
$$deny\ "SEJavaProgram" \xrightarrow{"*"\ invoke\ "<init>(ClassLoader)V"} "ClassLoader",$$
$$deny\ "SEJavaProgram" \xrightarrow{"*"\ invoke\ "<init>(*)V"} "URLClassLoader",$$
$$deny\ "SEJavaProgram" \xrightarrow{"*"\ invoke\ "<init>(*)V"} "SecureClassLoader",$$
$$deny\ "SEJavaProgram" \xrightarrow{"*"\ invoke\ "setSecurityManager(java.lang.SecurityManager)V"} "java.lang.System"$$

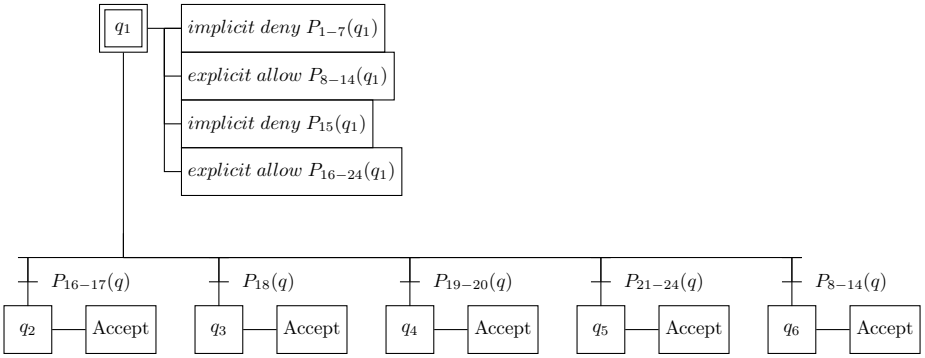
Le résultat correspond alors à une politique de sécurité exprimée dans notre langage de contrôle des relations et valide uniquement pour les objets de type *SEJavaProgram*. Sa particularité est qu'elle découle directement d'une transformation de la politique JAAS calculée depuis le *domaine de protection* relatif aux applets Java. En conséquence, cette politique est réutilisable pour tout objet de ce *domaine de protection*. Par exemple, une applet Java malicieuse ne pourra pas créer un nouveau classloader pour injecter des classes dans le noyau de la JVM car les cinq dernières règles l'y interdit ce qui est conforme aux objectifs de sécurité des applets (cf. figure 4.24).

### Transformation des règles de contrôle en règles DTE

Si les automates permettent effectivement de transcrire les objectifs de sécurité JAAS en règles de contrôle, nous devons néanmoins y adapter l'approche *Domain & Type Enforcement* pour obtenir une politique exploitable au format DTE. Cela constitue l'étape suivante de réécriture d'une politique JAAS en une politique DTE. Or, en accord avec notre projection du *Domain & Type Enforcement* sur les systèmes à objets établie en section 4.3.1, nous pouvons laisser l'expression des permissions telle qu'elle. De fait, il ne reste plus qu'à exprimer les entités *Sujet* et *Objet* à l'aide de types de sécurité :




(a) Automate reflétant la politique JAAS




(b) Automate reflétant les objectifs de sécurité

$$
\begin{aligned}
Q \equiv\{ & \\
& q_1 = \text{"}SEJavaProgram\text{"}, \quad q_2 = \text{"}java.lang.Class\text{"}, \\
& q_3 = \text{"}java.beans.Beans\text{"}, \quad q_4 = \text{"}java.beans.Introspector\text{"}, \\
& q_5 = \text{"}java.beans.PropertyEditorManager\text{"}, \quad q_6 = \text{"}java.lang.System\text{"}, \\
& q_7 = \text{"}java.awt.Robot\text{"}, \quad q_8 = \text{"}java.awt.Graphics2d\text{"}, \\
& q_9 = \text{"}java.lang.ClassLoader\text{"}, \quad q_{10} = \text{"}java.net.URLClassLoader\text{"}, \\
& q_{11} = \text{"}java.security.SecureClassLoader\text{"} \\
\} & \\
P \equiv\{ & \\
& P_1(q_1), \quad P_2(q_1), \quad P_3(q_1), \quad P_4(q_1), \quad P_5(q_1), \quad P_6(q_1), \\
& P_7(q_1), \quad P_8(q_1), \quad P_9(q_1), \quad P_{10}(q_1), \quad P_{11}(q_1), \quad P_{12}(q_1), \\
& P_{13}(q_1), \quad P_{14}(q_1), \quad P_{15}(q_1), \quad P_{16}(q_1), \quad P_{17}(q_1), \quad P_{18}(q_1), \\
& P_{19}(q_1), \quad P_{20}(q_1), \quad P_{21}(q_1), \quad P_{22}(q_1), \quad P_{23}(q_1), \quad P_{24}(q_1) \\
\} & \\
C \equiv\{ & \\
& C_1(q_1) = q_1 \; jaas.capability(\text{"}RuntimePermission\text{"}, \text{"}accessClassInPackage.*\text{"}), \\
& C_2(q_1) = q_1 \; jaas.capability(\text{"}PropertyPermission\text{"}, \text{"}*\text{"}, \text{"}read\text{"}), \\
& C_3(q_1) = q_1 \; jaas.capability(\text{"}AWTPermission\text{"}, \text{"}createRobot\text{"}), \\
& C_4(q_1) = q_1 \; jaas.capability(\text{"}AWTPermission\text{"}, \text{"}readDisplayPixels\text{"}), \\
& C_5(q_1) = q_1 \; jaas.capability(\text{"}RuntimePermission\text{"}, \text{"}createClassloader\text{"}), \\
& C_6(q_1) = q_1 \; jaas.capability(\text{"}RuntimePermission\text{"}, \text{"}setSecurityManager\text{"}) \\
\} &
\end{aligned}
$$

FIGURE 4.26 – Automates de contrôle du *domaine de protection* relatifs aux applets Java

— Soit au niveau modèle de manière à conserver une représentation du contexte de sécurité JAAS après la transformation par automate. Cela implique alors une politique DTE globale pour laquelle nous devons utiliser deux types de sécurité par entité - un pour JAAS, un pour notre langage - afin d'éviter tout conflit de règles.

— Ou bien, au niveau contrôle, de sorte à accélérer le processus de décision en réduisant la complexité des étiquettes de sécurité au seul type déduit de la signature de l'objet. Mais la perte d'information liée à l'absence du contexte JAAS sur lequel s'applique la politique calculée nécessite de maintenir une politique DTE par objet au prix d'une consommation mémoire accrue.

Pour notre part, nous préférons travailler directement sur les objectifs de sécurité plutôt que sur leur transcription en règles de contrôle. Cela signifie que nous devons calculer une étiquette de sécurité pour le contexte de sécurité Java étendu (cf. suggestion 4.3) qui nous sert de passerelle entre le langage JAAS et le nôtre. Ainsi, nous devons calculer une étiquette DTE déduite du contexte JAAS (cf. notation 4.2) et une étiquette DTE déduite de la signature de l'objet (cf. section 4.3.1). Nous appellerons cette approche *Security Enhanced Java* (SEJava) par analogie avec l'approche *Security Enhanced Linux* (SELinux) :

$$\overbrace{user\_u : role\_r}^{RBAC} : type\_t \qquad \overbrace{domain\_d}^{JAAS} : type\_j$$

(a) Étiquette de sécurité SELinux (b) Étiquette de sécurité SEJava

FIGURE 4.27 – Comparaison des étiquettes DTE entre *Security Enhanced Linux* et *Security Enhanced Java*

Nous pouvons alors remplacer les signatures utilisées dans les automates par des étiquettes SEJava et ainsi obtenir une politique de sécurité au format DTE. Or, un *domaine de protection* JAAS regroupe par définition plusieurs objets de la JVM et les entités *Objet* des capacités JAAS restent constantes quel que soit leur domaine de protection. Par déduction, nous avons seulement besoin de définir un type de sécurité pour le domaine de l'entité *Sujet* et un type de sécurité pour les classes protégées par JAAS. Dans notre exemple de l'applet Java, cela revient à labelliser le *domaine de protection* des applets par le type *applet_d* et à définir un type de sécurité particulier pour les classes protégées par JAAS (cf. annexe A) ce qui revient à réécrire la politique sous la forme suivante :

$$allow\ "applet\_d/*\_j" \xrightarrow{"*"\ invoke\ "getClasses()java.lang.Class[]"} "*\_d/class\_j",$$

$$allow\ "applet\_d/*\_j" \xrightarrow{"*"\ invoke\ "getFields()java.lang.reflect.Field[]"} "*\_d/class\_j",$$

$$allow\ "applet\_d/*\_j" \xrightarrow{"*"\ invoke\ "getMethods()java.lang.reflect.Method[]"} "*\_d/class\_j",$$

$$allow\ "applet\_d/*\_j" \xrightarrow{"*"\ invoke\ "getConstructors()java.lang.reflect.Constructor[]"} "*\_d/class\_j",$$

$$allow\ "applet\_d/*\_j" \xrightarrow{"*"\ invoke\ "getField(java.lang.String)java.lang.reflect.Field"} "*\_d/class\_j",$$

$$allow\ "applet\_d/*\_j" \xrightarrow{"*"\ invoke\ "getMethod(*)java.lang.reflect.Method"} "*\_d/class\_j",$$

$$allow\ "applet\_d/*\_j" \xrightarrow{"*"\ invoke\ "getConstructor(*)java.lang.reflect.Constructor"} "*\_d/class\_j",$$

$$allow\ "applet\_d/*\_j" \xrightarrow{"*"\ invoke\ "setDesignTime(Z)V"} "*\_d/jass\_beans\_j",$$

$$allow\ "applet\_d/*\_j" \xrightarrow{"*"\ invoke\ "setGuiAvailable(Z)V"} "*\_d/jass\_beans\_j",$$

$$allow\ "applet\_d/*\_j" \xrightarrow{"*"\ invoke\ "setBeanInfoSearchPath(String[])V"} "*\_d/jass\_introspector\_j",$$

$$allow\ "applet\_d/*\_j" \xrightarrow{"*"\ invoke\ "registerEditor(Class,Class)V"} "*\_d/jass\_propertyeditormanager\_j",$$

$$allow\ "applet\_d/*\_j" \xrightarrow{"*"\ invoke\ "setEditorSearchPath(String[])V"} "*\_d/jass\_propertyeditormanager\_j",$$

$$allow\ "applet\_d/*\_j" \xrightarrow{"*"\ invoke\ "getProperties()Properties"} "*\_d/jass\_system\_j",$$

$$allow\ "applet\_d/*\_j" \xrightarrow{"*"\ invoke\ "setProperties(Properties)V"} "*\_d/jass\_system\_j",$$

$$allow\ "applet\_d/*\_j" \xrightarrow{"*"\ invoke\ "getProperty(String)String"} "*\_d/jass\_system\_j",$$

$$allow\ "applet\_d/*\_j" \xrightarrow{"*"\ invoke\ "getProperty(String,String)String"} "*\_d/jass\_system\_j",$$

$$deny\ "applet\_d/*\_j" \xrightarrow{"*"\ invoke\ "<init>()V"} "*\_d/jass\_robot\_j",$$

$$deny\ "applet\_d/*\_j" \xrightarrow{"*"\ invoke\ "<init>(java.awt.GraphicsDevice)V"} "*\_d/jass\_robot\_j",$$

$$deny\ "applet\_d/*\_j" \xrightarrow{"*"\ invoke\ "setComposite(java.awt.Composite)V"} "*\_d/jass\_graphics2d\_j",$$

$$deny\ "applet\_d/*\_j" \xrightarrow{"*"\ invoke\ "<init>()V"} "*\_d/classloader\_j",$$

$$deny\ "applet\_d/*\_j" \xrightarrow{"*"\ invoke\ "<init>(ClassLoader)V"} "*\_d/classloader\_j",$$

$$deny\ "applet\_d/*\_j" \xrightarrow{"*"\ invoke\ "<init>(*)V"} "*\_d/classloader\_j",$$

$$deny\ "applet\_d/*\_j" \xrightarrow{"*"\ invoke\ "<init>(*)V"} "*\_d/classloader\_j",$$

$$deny\ "applet\_d/*\_j" \xrightarrow{"*"\ invoke\ "setSecurityManager(java.lang.SecurityManager)V"} "*\_d/system\_j"$$

Nous pouvons immédiatement remarquer que le passage au langage DTE crée des doublons de règles, notamment sur les dernières de la politique. En effet, le type de sécurité "classloader_j" associé à la classe *Classloader* se propage automatiquement aux classes *URLClassloader* et *SecureClassloader* du fait d'une relation d'héritage entre celles-ci. En supprimant ces doublons nous pouvons réduire d'autant la taille de la politique.

Au final, nous sommes donc partis d'un objet de la JVM pour accéder à son *domaine de protection* JAAS et avons réécrit dynamiquement les privilèges de ce domaine en une politique de sécurité au format DTE. Parce qu'un domaine JAAS regroupe plusieurs objets, la politique ainsi obtenue est valide pour tous les objets de ce domaine. La dernière étape consiste alors à faire appliquer cette politique par un moniteur de référence adapté. Nous verrons une application pratique de cette méthode dans le chapitre suivant.

## 4.4 Conclusion

Par ce chapitre, nous avons défini une méthode de contrôle complémentaire à JAAS conservant les points positifs de ce dernier. En effet, le fait que la logique

de contrôle de JAAS soit centrée sur les seuls objets importants du système, est pertinent pour éliminer le bruit de nos politiques et ainsi se concentrer sur l'essentiel. Cependant, il s'avère qu'il existe des faiblesses dans l'implémentation de JAAS que les malwares Java s'empressent d'exploiter pour contourner cette logique de contrôle comme le montre notre analyse du modèle d'attaque. Dans l'état actuel des choses, JAAS est donc de moins en moins capable d'adresser les futures menaces car c'est son implémentation qui est attaquée.

Nous avons donc défini la possibilité de mettre en œuvre une logique de contrôle par automate pour servir de "garde fou" à JAAS et donc renforcer son implémentation dans l'éventualité d'une attaque sur celle-ci. Il s'agit là de l'expression même d'une défense en profondeur car un malware Java devra non-seulement contourner JAAS, mais il devra échapper également à la vigilance de nos automates ce qui n'est pas impossible [17] mais peu probable dans l'état actuel des connaissances en matière d'exploitation de vulnérabilités Java. La première constatation de cette approche est que nos automates font au moins aussi bien que JAAS et vont même au delà en gérant les élévations de privilèges de façon plus rigoureuse.

En vue d'une implémentation de cette logique de contrôle par automates, nous avons proposé de créer automatiquement une politique dans l'esprit du *Domain & Type Enforcement*. L'idée était alors d'utiliser une technique de contrôle obligatoire alimentée par des automates qui calculent des politiques dynamiques. Il en résulte que la transformation d'objectifs de sécurité en règles DTE est assez directe puisque nos automates génèrent déjà des règles de contrôle compatibles avec ce langage. Nous sommes notamment capables de calculer des types de sécurité à partir des signatures d'objets pour être compatible avec la philosophie du *Domain & Type Enforcement* et de bénéficier de ses avantages dont la factorisation de règles.

Ainsi, si nos automates prennent en entrée des objectifs de sécurité pour les transformer dynamiquement en règles de contrôle, la question était de connaître les objectifs de sécurité d'une application voir de la JVM. Nous proposons que ces objectifs de sécurité soient ceux de JAAS car l'analyse de notre modèle d'attaque a montré qu'un malware Java violait toujours une permission JAAS pour réaliser une élévation de privilèges. Pour cela, nous transformons chaque permission JAAS en politique DTE.

De fait, nous nous appuyons sur les connaissances de JAAS rassemblées dans un objet particulier appelé *domaine de protection* afin de faire le lien entre le langage de contrôle JAAS et notre langage. Ainsi, en définissant un contexte de sécurité "à mi-chemin" entre ces deux langages, et en nous appuyant sur les spécifications techniques des capacités JAAS, nous pouvons retrouver des triplets *(Sujet, Objet, Permission)* qui sont à la base de notre langage de contrôle. Grâce à cela, nous pouvons alors modéliser les objectifs de sécurité des objets Java dont

17. De façon générale, on considère que le risque "zéro" n'existe pas

on a extrait le *domaine de protection* ; l'étape finale consistant à projeter ces objectifs vers le langage du *Domain & Type Enforcement* et les transformer en règles de contrôle applicables.

Au final, nous disposons donc d'une logique de contrôle plus avancée que celle de JAAS sans pour autant lui faire concurrence. Mais nous utilisons un langage simple qui est celui des capacités JAAS pour les transformer automatiquement en une politique de contrôle DTE, qui en pratique serait trop difficile à définir à la main. Le seul prérequis à cela est de fournir une retranscription de ces capacités en un ensemble de permissions exprimées dans notre langage de contrôle. Cette retranscription donne alors lieu à plusieurs automates qui ont pour particularité d'être calculés dynamiquement et d'être réutilisables tant que les spécifications JAAS n'évoluent pas.

| | Efficacité | Passage à l'échelle | Maintenabilité |
|---|---|---|---|
| JAAS | ++ | +++ | ++ |
| Analyse de programme | - - - | - | ++ |
| Coloration dynamique | + | - - | + |
| Approche formelle | ++ | - - - | - - |
| Contrôle d'accès DTE | ++ | +++ | - - |
| Contrôle par automate | +++ ① | +++ ② | +++ ③ |

TABLE 4.15 – Comparaison de notre logique de contrôle par automate avec les approches alternatives (cf. tableau 2.2 en conclusion de l'état de l'art).

① Nous avons montré en section 4.1 que nous maîtrisions mieux les élévations de privilèges que JAAS mais surtout que nous étions en mesure d'adresser la totalité du modèle d'attaque établi ; ② Nous bénéficions ici de la propriété de JAAS qui est d'appliquer un contrôle sur les seuls objets pertinents ; ③ la maintenabilité est assurée par une traduction automatique des règles JAAS en règles DTE. Nous disposons cependant d'un avantage sur JAAS car nous avons seulement besoin de connaître les spécifications des capacités sans pour autant être intrusif alors que JAAS impose de modifier le code des applications Java pour garantir l'application de sa politique.

Chapitre 5

# Implémentation de Security Enhanced Java

## Sommaire



> *Le problème avec les langages orientés objets est cet environnent implicite qu'ils apportent avec eux. Vous vouliez une banane mais vous avez eu un gorille tenant une banane et toute la jungle autour.*
>
> John Armstrong [94]

Nous avons commencé cette thèse en expliquant dans le chapitre 2 qu'il n'existait pas de protection obligatoire satisfaisante pour les environnements Java.

Plus particulièrement, les alternatives disponibles abordent le problème en tenant compte principalement des aspects langage de programmation et environnement d'exécution de Java alors même qu'il s'agit avant tout d'un système à objets. C'est donc pour être en rupture avec cette vision réductrice sur Java que nous avons réalisé une modélisation des systèmes à objets dans le chapitre 3. Le modèle que nous proposons est suffisamment général pour supporter des langages au-delà des langages à objets ce qui permet de l'appliquer à tout système qui décrit des objets même si le langage de ce système n'est pas orienté objet.

Grâce à cette modélisation nous avons pu établir qu'en contrôlant les relations entre objets du système, nous pouvions, non seulement garantir l'application de politique de sécurité, mais surtout offrir une logique de contrôle plus automatisée que de simplement comparer un *Sujet*, un *Objet* et une *Permission* à un ensemble de règles de contrôle. Nous avons décrit et formalisé cette logique de contrôle par automates dans le chapitre 4 où celle-ci s'avère particulièrement adaptée pour maîtriser les élévations de privilèges ; aspect sur lequel JAAS, le modèle de sécurité standard de Java, se révèle peu efficace. Nous utilisons ces automates pour générer automatiquement des règles de contrôle à partir d'objectifs de sécurité plus abstraits. Dans ce but, nous avons décrit une méthode pour transformer les capacités JAAS données aux objets de la JVM en objectifs de sécurité exprimés dans notre langage de relations inter-objets. Ainsi, en utilisant une méthode qui nécessite de coopérer avec JAAS et la JVM, nous savons comment transformer une politique JAAS en une politique compatible *Domain & Type Enforcement* (DTE).

Bien entendu, ce que nous avons présenté jusqu'ici s'appuie sur des résultats expérimentaux obtenus tout au long de ces années de thèse. C'est pourquoi, nous proposons dans ce chapitre les éléments techniques que nous avons utilisés pour instrumenter une machine virtuelle Java et déployer une approche DTE pour contrôler les objets Java. Nous terminerons ce chapitre en présentant les résultats que nous avons obtenus en traduisant automatiquement les politiques Java en politiques DTE.

La figure 5.1 montre l'architecture de notre prototype d'expérimentation. Celui-ci se compose d'une bibliothèque qui implémente les primitives nécessaires au contrôle des objets Java et d'une bibliothèque d'instrumentation pour dialoguer avec la machine virtuelle Java du projet OpenJDK [1]. Ces deux composants sont liés par une interface de programmation C/C++ qui permet de réutiliser notre bibliothèque logique dans un environnent autre que OpenJDK tel que Dalvik, la JVM d'Android.

## 5.1 Instrumentation d'une machine virtuelle Java

Comme expliqué précédemment, Java implémente un système à objets basé sur les classes. Sa machine virtuelle met en œuvre un système entièrement virtualisé.

1. `http://openjdk.java.net`

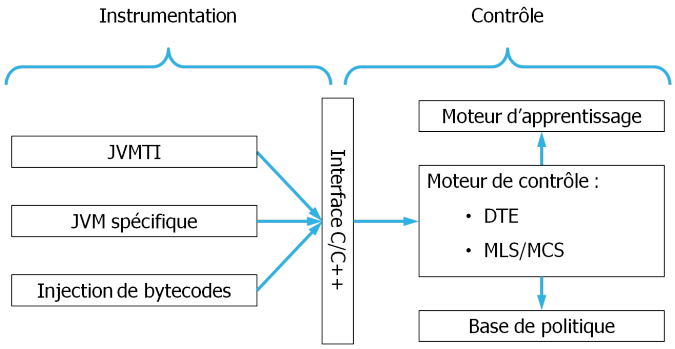



FIGURE 5.1 – Architecture fonctionnelle de Security Enhanced Java

La première possibilité sera donc de pénétrer dans le cœur de la JVM pour obtenir un accès sans contraintes à ce système. L'ensemble des relations possibles à contrôler dépendra alors pour beaucoup des capacités de notre instrumentation. En effet, notre hypothèse 3.1 établit que nous ne savons contrôler que les seules relations observables, donc ce contrôle et la garantie des politiques dépendront de ce que nous pourrons observer. En pratique, les relations d'interactions sont les plus faciles à observer car elles ne font pas intervenir de l'inspection de données. C'est pourquoi, nous nous concentrerons majoritairement sur celles-ci.

Par ailleurs, nous devons développer un moniteur de référence qui respecte les critères de Anderson [21]. C'est à dire un composant capable d'intercepter les relations inter-objets de façon transparente tout en étant protégé des influences de ces derniers. Le troisième critère est celui de la validation formelle du moniteur de référence par rapport au modèle de contrôle implémenté. Le tableau 5.1 présente notre appréciation sur trois méthodes d'instrumentation que nous avons évaluées selon deux critères principaux : L'intégration au sein de la JVM (facilité d'accès aux composants internes, coût sur les performances, ...) et la facilité de mise en œuvre (effort d'ingénierie nécessaire, compatibilité avec d'autres machines virtuelles, ...).

| | Niveau d'intégration | Mise en œuvre |
|---|---|---|
| Modification d'une JVM | + + + | + |
| Instrumentation JVMTI | + + | + + |
| Injection de bytecodes | + | + + + |

TABLE 5.1 – Comparatif de trois méthodes d'instrumentation d'une machine virtuelle Java

### 5.1.1 Augmentation d'une JVM existante

L'approche intuitive consiste à utiliser une machine virtuelle Java dont le code source est disponible telles que OpenJDK [2] ou Avian [3]. Pour information, La

2. `http://openjdk.java.net`
3. `http://oss.readytalk.com/avian/`

version française de Wikipédia [95] propose un tableau comparatif des différentes JVM disponibles. Sans pour autant garantir l'exactitude des informations qui y sont présentées, ce comparatif constitue un point de départ pour toute personne désireuse d'évaluer ce type d'instrumentation.

Dans cette section, nous ne donnerons pas de détails techniques tant l'implémentation d'une machine virtuelle Java peut différer d'une version à l'autre. Toutefois nous nous efforcerons de donner un retour d'expériences sur nos différentes tentatives de modification d'une JVM existante.

**OpenJDK**

Ainsi, notre premier choix s'est porté sur OpenJDK 1.7 qui est la JVM de référence, c'est à dire celle qui possède la meilleure compatibilité avec les spécifications Java. La difficulté majeure est qu'elle est constituée de plusieurs centaines de milliers de lignes de code en grande majorité non commentées, le tout agrémenté d'un faible support communautaire. Ce choix impose donc un effort significatif de rétro-ingénierie.

Autre aspect, une JVM implémente généralement deux types de processeurs Java. Le premier réalise une exécution interprétée des bytecodes Java ce qui autorise des techniques d'optimisation avancées telles que *l'exécution dans le désordre* [4]. Le second processeur pratique quant à lui la compilation à la volée de sorte à exécuter le code Java directement en langage natif et ainsi accélérer l'exécution du programme. La particularité d'*Hotspot*, le nom de la machine virtuelle d'OpenJDK, est qu'elle utilise un mode d'exécution hybride où une méthode Java souvent appelée aura plus tendance à être compilée qu'une méthode rarement utilisée.

En d'autres termes, pour intercepter les appels de méthodes il est nécessaire de modifier l'implémentation de ces deux processeurs en faisant attention à ne pas perturber le fonctionnement des différents mécanismes d'optimisation et de protection. Nous attestons que cela est réalisable mais au prix d'un effort considérable qu'il serait inutile de détailler ici. Pour mémoire, nous rappelons que la complexité de fonctionnement d'OpenJDK est parfois à l'origine de vulnérabilités comme l'atteste la CVE-2012-1723 que nous avions présentée dans la section 4.2.2 relative à l'étude du modèle d'attaque.

4. Ce paradigme a pour but de rentabiliser au mieux les ressources du système à chaque cycle d'exécution.

### Avian

Comme expliqué dans l'état de l'art, Java s'appuie sur une machine virtuelle qui s'exécute au-dessus d'un hyperviseur. Théoriquement, ce dernier peut exécuter plusieurs machines virtuelles en parallèle [96] même si, à notre connaissance, seul Android en est capable en pratique. La machine virtuelle *Avian* s'appuie sur cette particularité pour s'exécuter au-dessus d'un hyperviseur Java sur étagère, en l'occurrence celui d'OpenJDK et d'Android, afin de se poser comme une alternative à *Hotspot* et *Dalvik*. Le principal avantage d'*Avian* est bien évidement une portabilité garantie sur différentes architectures, aussi bien fixes que mobiles.

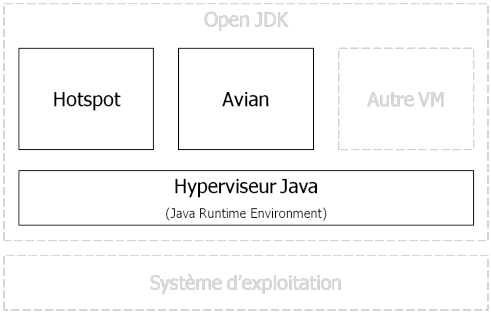



FIGURE 5.2 – Architecture fonctionnelle de OpenJDK

La philosophie de cette machine virtuelle est de limiter son code source au strict nécessaire de sorte à consacrer un maximum de ressources à l'exécution du programme plutôt qu'à son optimisation. Il s'agit ici clairement d'un avantage pour nous car un code source réduit facilite son analyse et l'intégration de modules tiers. Malheureusement, cette simplicité fait que Avian ne respecte pas parfaitement les spécifications Java et notamment celles portant sur JAAS. Ainsi, nous avons relevé quelques dysfonctionnements relatifs au calcul des contextes de sécurité JAAS qui ne nous ont pas permis de pousser plus en avant nos expérimentations.

Par contre, la communauté Avian semble plutôt active et à l'heure où nous écrivons ces lignes, une nouvelle version est disponible et pour laquelle de nombreux bugs semblent avoir été corrigés. Nous ne serions que conseiller de retenter l'expérience avec cette machine virtuelle.

### Dalvik

*Dalvik* est la machine virtuelle Java de Android qui a été remplacée récemment par *ART*, un runtime d'exécution similaire à *.Net*. Nous nous sommes intéressés à *Dalvik* car nous voulions évaluer notre approche dans un contexte éloigné des machines virtuelles Java traditionnelles. Ce travail présenté dans [80] confirme la portabilité de notre approche.

#### Discussion

Le fait de travailler à un niveau aussi bas de la machine virtuelle permet d'accéder facilement aux organes internes et autorise de fait une bonne intégration de la mesure de protection au sein de Java. De plus, en se branchant directement au cœur de l'interpréteur Java nous réduisons mécaniquement les temps de latence entre l'observation d'un appel de méthode et la prise de décision.

Toutefois, le choix de modifier une machine virtuelle Java implique des étapes fastidieuses de rétro-ingénierie et de développement de code. Bien qu'il soit possible de trouver des machines virtuelles légères, cela doit tout de même s'inscrire dans une stratégie à long terme sous réserve d'une coopération avec la communauté si elle existe. Notamment pour un industriel, le fait de soumettre un standard Java [5] sur le thème d'un mécanisme de protection obligatoire apporterait un plus stratégique non négligeable. En effet, si le standard est accepté alors la communauté intégrera et maintiendra le dit standard dans les futures versions de Java, répartissant ainsi les efforts d'ingénierie requis.

### 5.1.2 Utilisation d'une interface standard

On comprend assez vite que modifier une machine virtuelle Java existante pour l'adapter à nos besoins est plutôt difficile. Or, certaines implémentations proposent en standard une interface de programmation pour interroger le système à objets et en connaître son état ; l'usage premier étant bien entendu le debug et le profiling d'applications. La plus notable d'entre elles est la *Java Virtual Machine Tool Interface* [97].

#### JVMTI

Il s'agit d'une API Java standard qui s'inscrit en remplacement du standard *Java Platform Debugger Architecture* [98]. L'idée est de créer une bibliothèque de type "agent" qui sera chargée au démarrage de la machine virtuelle. Cet agent doit ensuite s'inscrire à un ensemble d'événements internes de la JVM (allocation d'objet, verrouillage, création de fil d'exécution, etc.) et présenter une fonction de *callback* pour chacun d'eux. La machine virtuelle émet ensuite des événements relatifs aux changements d'état du système pendant l'exécution de l'application que ces agents interceptent et traitent. La nature et le type d'événement qu'un agent peut traiter diffèrent si celui-ci est écrit en Java ou en code natif C/C++. [99] donne un tutoriel pour écrire un agent JVMTI natif.

5. `https://jcp.org/en/home/index`

Ainsi, l'événement qui nous intéresse est celui levé par l'interpréteur Java pour signaler un appel de méthode : `METHOD_ENTRY`. Selon la documentation officielle [97], le callback correspondant reçoit une référence sur le thread Java qui réalise l'appel ainsi qu'une référence sur la méthode appelée :

$$
\begin{aligned}
&voidJNICALL\\
&\quad MethodEntry(jvmtiEnv * jvmti,\\
&\qquad\qquad JNIEnv * jni,\\
&\qquad\qquad jthread\ thread,\\
&\qquad\qquad jmethodID\ method)
\end{aligned}
$$

Or, pour appliquer notre méthode de contrôle, nous avons besoin de connaître également la méthode appelante ainsi que les objets appelants et appelés. Pour cela, nous devons inspecter la pile d'exécution du thread en utilisant les fonctions *GetStackTrace* et *GetLocalInstance* de JVMTI. Ensuite, nous devons obtenir les signatures de ces quatre objets grâce aux fonctions *GetClassSignature* et *GetMethodName* de JVMTI. La figure 5.3 donne une vision globale du fonctionnement de notre agent JVMTI.

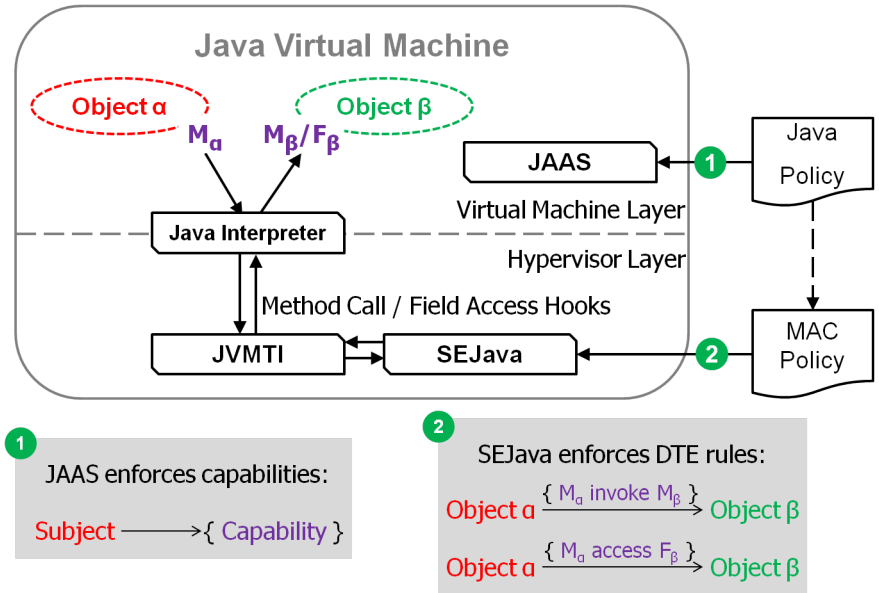



FIGURE 5.3 – Intégration de SEJava dans OpenJDK via un agent JVMTI

L'agent JVMTI a pour objectif d'intercepter les appels entre une méthode $M_\alpha$ et $M_\beta$ ainsi que les accès au champ $F_\beta$. Les fonctions de l'API JVMTI nous permettent facilement de retrouver les contextes de sécurité des objets $\alpha$ et $\beta$ ainsi que la signature des membres concernés. Grâce à notre approche par automates, la politique DTE appliquée peut être mise à jour automatiquement à partir des privilèges JAAS donnés à chaque objet selon la méthode décrite en section 4.3.2.

En soit cette approche fonctionne très bien comme l'atteste notre article [81] mais elle souffre de deux limitations intrinsèques. D'abord, les fonctions JVMTI

mentionnées précédemment imposent à la JVM d'annuler ses optimisations afin de retrouver les informations demandées et plus particulièrement lorsqu'il s'agit de reconstruire la pile d'exécution. De plus, ces informations ne sont jamais retournées directement, les concepteurs de JVMTI préférant renvoyer une copie comme l'atteste, par exemple, le code source de la fonction *GetMethodName* (cf. lignes 16, 23 et 36 du listing 5.4). Ces deux raisons font que les échanges entre la JVM et un agent JVMTI ne sont pas aussi rapides qu'ils pourraient l'être.

```
1  // method_oop - pre-checked for validity, but may be NULL meaning obsolete
       method
2  // name_ptr - NULL is a valid value, must be checked
3  // signature_ptr - NULL is a valid value, must be checked
4  // generic_ptr - NULL is a valid value, must be checked
5  jvmtiError
6  JvmtiEnv::GetMethodName(methodOop method_oop, char** name_ptr, char**
       signature_ptr, char** generic_ptr) {
7    NULL_CHECK(method_oop, JVMTI_ERROR_INVALID_METHODID);
8    JavaThread* current_thread  = JavaThread::current();
9
10   ResourceMark rm(current_thread); // get the utf8 name and signature
11   if (name_ptr == NULL) {
12     // just don't return the name
13   } else {
14     const char* utf8_name = (const char *) method_oop->name()->as_utf8();
15     *name_ptr = (char *) jvmtiMalloc(strlen(utf8_name)+1);
16     strcpy(*name_ptr, utf8_name);
17   }
18   if (signature_ptr == NULL) {
19     // just don't return the signature
20   } else {
21     const char* utf8_signature = (const char *) method_oop->signature()->
         as_utf8();
22     *signature_ptr = (char *) jvmtiMalloc(strlen(utf8_signature) + 1);
23     strcpy(*signature_ptr, utf8_signature);
24   }
25
26   if (generic_ptr != NULL) {
27     *generic_ptr = NULL;
28     Symbol* soop = method_oop->generic_signature();
29     if (soop != NULL) {
30       const char* gen_sig = soop->as_C_string();
31       if (gen_sig != NULL) {
32         jvmtiError err = allocate(strlen(gen_sig) + 1, (unsigned char **)
             generic_ptr);
33         if (err != JVMTI_ERROR_NONE) {
34           return err;
35         }
36         strcpy(*generic_ptr, gen_sig);
37       }
38     }
39   }
40   return JVMTI_ERROR_NONE;
41 } /* end GetMethodName */
```

FIGURE 5.4 – Code source de la fonction JVMTI *GetMethodName*.

En conséquence, JVMTI convient très bien pour une preuve de concept d'un moniteur de référence, mais celui-ci n'a jamais été conçu pour cet usage. En effet, l'approche réduit considérablement les performances globales de la JVM lorsque nous utilisons JVMTI. La solution serait alors d'ajouter au standard JVMTI des primitives de sécurité et ainsi fournir un accès direct aux données dont nous avons besoin sans la lourde mécanique des fonctions standards.

### Augmentation d'une implémentation JVMTI

L'un des rares documents techniques disponibles au sujet de JVMTI [100] donne une vision globale de son organisation interne. On y découvre notamment que son implémentation repose en grande partie sur du code généré à partir de schémas XML. L'idée sera donc de déclarer nos primitives de contrôle dans le modèle de l'interface (jvmti.xml) puis de régénérer JVMTI. Le résultat correspond alors à un fichier source et une entête C++ (jvmtiEnv.cpp/.h) à placer dans le code source de OpenJDK et contenant le prototype de nos fonctions.

Cette approche nous permet alors de bénéficier du degré d'intégration de JVMTI dans la JVM sans en perturber son fonctionnement. Il s'agit là d'une approche à cheval entre modifier soit même une JVM et utiliser une interface d'instrumentation standard. Cependant, nous n'avons pas exploré plus en avant cette idée par manque de temps et de ressources.

### Discussions

Au final JVMTI constitue une bonne option d'instrumentation de JVM du fait que l'interface de programmation est totalement standard et portable. Néanmoins on touche rapidement aux limites des capacités de JVMTI lorsqu'il s'agit d'accéder à des informations très précises du système comme par exemple les *domaine de protection* JAAS qui sont pour nous indispensables pour générer automatiquement les politiques DTE.

En effet, dialoguer avec JAAS impose d'exécuter du code Java ne serait-ce que pour extraire les contextes de sécurité JAAS dans un format exploitable. De fait, parce que notre agent JVMTI traite tous les appels de méthodes, on tombe rapidement dans des situations où cet agent intercepte ses propres appels de méthodes. En guise de palliatif nous avons utilisé un compteur par thread pour détecter les appels récursifs à l'agent mais cela ne nous a pas donné entière satisfaction.

En réalité il s'avère qu'intercepter tous les appels de méthodes de façon exhaustive est problématique car on se retrouve bien souvent à contrôler beaucoup de bruit alors que le coût du dialogue avec la JVM est grand. C'est donc majoritairement pour cette raison que l'overhead que nous avons constaté dans nos expériences est aussi considérable (+600% en moyenne). Nous n'avons donc pas exploré plus en avant l'option JVMTI.

### 5.1.3 Interception dynamique d'appels de méthode

JVMTI intercepte donc tous les appels de méthodes sans possibilité de discriminer les seuls appels vraiment pertinents à contrôler. Notre idée est alors de mimer JAAS en interceptant les seuls appels pour lesquels nous savons qu'il existe une règle de contrôle d'accès, tous les autres étant alors ignorés puisqu'ils représentent du bruit par rapport à la politique de sécurité considérée. Dans ce but, nous nous sommes inspirés de [86] qui présente une méthode générique pour intercepter les appels à n'importe quelle fonction d'un binaire Windows. L'idée consiste à remplacer les premiers octets de la fonction à instrumenter par une instruction de saut (`jump`, `call`, ...) dans le but de détourner les appels vers une autre fonction avant de reprendre l'exécution du code original.

#### Injection de bytecodes Java

Ainsi, notre objectif sera d'injecter en entête des méthodes Java le bytecode *invokestatic* qui prend en opérande l'emplacement d'une méthode statique. Dans notre cas, nous utilisons une méthode statique mais native qui a pour rôle de servir de moniteur de référence [21]. Lorsque la méthode Java ainsi instrumentée est appelée, l'exécution est automatiquement détournée sur notre moniteur de référence depuis lequel nous pourrons retrouver les informations nécessaires à une prise de décision (objets appelant/appelé et méthodes appelante/appelée).

L'injection peut se faire depuis un agent JVMTI qui intercepte le chargement de classes Java via l'événement `CLASS_FILE_LOAD_HOOK` comme illustré sur la figure 5.5. Cet agent peut être écrit en C/C++ [6] mais nous avons préféré un agent Java afin de nous appuyer sur la bibliothèque *Javassist* [101] qui offre toutes les fonctionnalités nécessaires et pour laquelle il existe de nombreux tutoriels [102]. Pour les classes chargées avant l'agent JVMTI, il est nécessaire d'appeler la fonction *RetransformClasses* [103].

Contrairement à l'idée d'un agent JVMTI classique présentée en section 5.1.2, un tel moniteur de référence n'est appelé que lorsque cela s'avère nécessaire et en cela nous nous rapprochons du fonctionnement du moniteur de référence de JAAS. Mais contrairement à ce dernier qui nécessite l'accès au code source et la coopération du développeur, nous intervenons directement sur le binaire des classes Java avec juste une politique de sécurité qui détermine quelles sont les méthodes à protéger. Plus particulièrement, lorsque l'on intègre notre logique de contrôle par automates et la transcription des politiques JAAS en politiques DTE, toutes les méthodes listées par [20] s'en trouvent alors automatiquement instrumentées.

6. cf. implémentation de l'exemple `java_crw_demo` de OpenJDK (`http://hg.openjdk.java.net/jdk7/jdk7/jdk/file/tip/src/share/demo/jvmti/`)

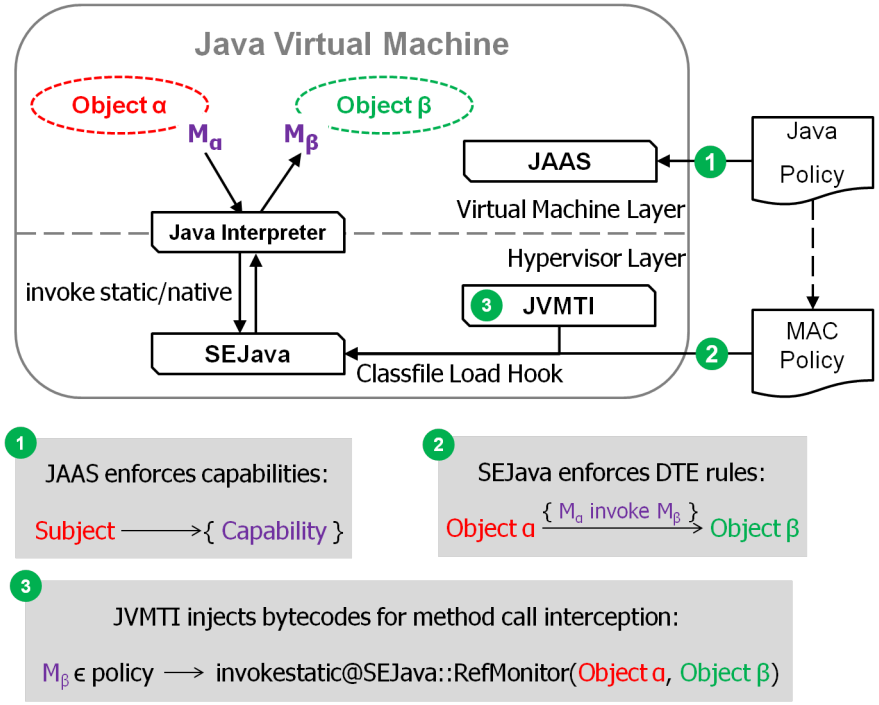



FIGURE 5.5 – Intégration de SEJava dans OpenJDK via de l'injection de bytecodes

L'agent JVMTI a pour objectif d'intercepter le chargement de classes dans la JVM. Pour chacune d'entre elles, la politique de sécurité détermine s'il est nécessaire d'instrumenter les méthodes que celles-ci définissent ou non. Lorsqu'une méthode $M_\beta$ est méthode source d'au moins une règle, l'API Javassist injecte un bytecode `invokestatic` en entête de cette méthode pour que l'interpréteur Java fasse suivre l'appel de méthode à contrôler à notre moniteur de référence.

Néanmoins cette technique d'instrumentation possède quelques limites et en premier lieu le contrôle de l'accès aux champs d'un objet. En effet, on ne peut injecter des bytecodes que dans le corps d'une méthode et il n'est donc pas possible d'instrumenter un champ de la même manière. Toutefois, il faut noter que la communauté Java considère que l'utilisation d'accesseurs est une bonne pratique de programmation lorsque le champ concerné revêt une importance particulière par rapport au reste du programme [104]. Sachant que les accesseurs Java sont implémentés sous la forme de méthodes, il sera tout de même possible de réaliser du contrôle "best-effort" sur ce type de relation.

La seconde limitation est que nous ne disposons pas de primitives pour retrouver rapidement et facilement les informations nécessaires à une prise de décision. C'est pourquoi, nous avons été contraints de nous appuyer sur JVMTI pour passer la JVM en debug et interroger le système à objets avec tous les problèmes de performance que cela entraîne. Cette constatation nous confirme une fois de plus la nécessité d'enrichir la JVM avec des fonctionnalités de contrôle d'accès bas niveau.

Enfin, la troisième limitation est plutôt un constat : si un agent JVMTI est en mesure d'injecter du code dans le corps d'une méthode alors un autre agent peut tout aussi bien enlever ce même code. Plus particulièrement, cette instrumentation peut être réalisée par du code Java et donc rien n'empêche un objet malicieux de défaire notre instrumentation pour contourner la protection. Il s'agit là d'une situation purement hypothétique mais JAAS qui suit une approche analogue avec son *SecurityManager* peut être désactivé de cette manière. Donc l'idée d'intercepter dynamiquement les appels de méthodes est bonne mais l'appliquer par de l'injection de bytecodes n'est pas réalisable si l'on souhaite respecter tous les critères du moniteur de référence de Anderson [21].

#### Marquage dynamique des méthodes depuis l'interpréteur Java

Si nous prenons un peu de recul, le fait d'injecter du code dans le corps d'une méthode correspond indirectement à "marquer" cette méthode pour le contrôle de son accès. Or, si nous avions la possibilité de marquer les méthodes à protéger directement depuis l'hyperviseur Java, alors il ne serait plus possible pour un objet malicieux de désactiver la protection. Cette idée a été explorée par [105] en modifiant la représentation mémoire des méthodes Java au sein de la JVM. Il s'agit d'ajouter un bit particulier aux objets internes *methodOop* de OpenJDK qui commande à l'interpréteur Java d'appeler notre moniteur de référence lorsqu'il est sur le point de réaliser un appel de méthode.

Fonctionnellement, le résultat est le même que précédemment sauf qu'il nécessite un effort d'ingénierie conséquent pour être mis en œuvre. Cependant, que ce soit par injection de bytecodes ou bien par marquage interne, nos expérimentations

ont montrées que nous garantissons l'application obligatoire de politiques de sécurité et maîtrisons les élévations de privilèges là où JAAS se révèle insuffisant. Mais la particularité de cette variante d'instrumentation est que le bénéfice obtenu, à savoir un véritable moniteur de référence de Anderson pour Java, vaut la peine d'investir dans le développement ou la modification d'une machine virtuelle Java.

### 5.1.4 Conclusion sur l'instrumentation

Par cette section nous avons voulu présenter les différentes techniques d'instrumentations que nous avons testées. Initialement, nous sommes partis sur l'idée d'augmenter une machine virtuelle Java existante. Mais au regard de la complexité de fonctionnement des organes internes d'une JVM [7], la tâche était trop compliquée pour être réalisée dans le temps imparti et peu valorisable d'un point de vue scientifique.

Le fait d'utiliser JVMTI nous a permis de tester notre approche de contrôle d'accès et d'obtenir rapidement des résultats concrets que nous présentons en fin de ce chapitre. Les problèmes de performances rencontrés nous ont empêché d'expérimenter des applications plus complexes qu'une applet Java comme par exemple un serveur d'applications OSGi.

Dans cette optique nous avons alors cherché à optimiser les temps de contrôle d'accès en réduisant notamment le nombre d'appels au moniteur de référence. Pour cela nous avons choisi d'utiliser l'injection de code pour détourner les seuls appels pertinents à contrôler. Mais en voulant améliorer l'approche par le développement d'un mécanisme d'interception des appels directement au cœur de la JVM, nous sommes revenus à l'idée initiale qui réclame un lourd tribut en matière d'ingénierie.

Au final, après avoir évalué plusieurs stratégies d'instrumentation et poussé certaines d'entre elles au delà de leurs possibilités, nous ne pouvons faire qu'un seul constat. En effet, un noyau de système d'exploitation propose nativement l'interception d'appels systèmes et la manipulation d'objets comme les processus ou les pages mémoires. Ce type de fonctionnalité bas niveau a permis entre autre de mettre au point l'architecture FLASK [106] qui est au cœur de *Security Enhanced Linux* notamment. Mais les hyperviseurs Java actuels ne fournissent ni la possibilité de développer facilement un module noyau, ni d'intégrer des mécanismes de sécurité tels que le nôtre ou même un JAAS 2.0, par exemple. Tout au plus, les éditeurs mettent à disposition des interfaces de debug comme JVMTI qui ne sont pas conçues pour cet usage. De fait, l'intégration d'un module de sécurité dans le noyau de la JVM passe obligatoirement par une utilisation détournée des outils actuellement disponibles. Il en résulte que s'il est

7. environ 800 000 lignes de code pour OpenJDK

effectivement possible d'appliquer un contrôle d'accès obligatoire à un système à objets, les machines virtuelles Java actuelles doivent évoluer vers un véritable noyau de système d'exploitation avec les facilités d'instrumentation. Cependant, nous verrons qu'en pratique la réécriture des politiques JAAS limite notablement le surcoût et rend plus intéressante l'idée d'une intégration au niveau d'un hyperviseur Java.

## 5.2 Intégration de la logique de contrôle

Le rôle du moniteur de référence est d'observer les relations entre les objets du système, formalisées selon notre modèle général. Mais la responsabilité d'autoriser ou non une relation dépend d'un autre composant que nous nommerons *moteur de décisions*. Cette section a pour but de présenter les éléments clefs pour implémenter et utiliser ce composant.

### 5.2.1 Interface de programmation applicative (API)

Parce que nous avons fait le choix de séparer l'implémentation en deux bibliothèques dynamiques, il est nécessaire de définir une interface de programmation pour que le moniteur de référence puisse dialoguer avec le moteur de décision. La difficulté est que le moniteur de référence peut être développé en C++ (OpenJDK), en C (Dalvik) ou tout autre langage. Il est donc nécessaire de définir une interface qui peut facilement s'adapter à ces environnements de programmation afin de permettre la réutilisation du moteur de décision dans un autre contexte que celui de Java.

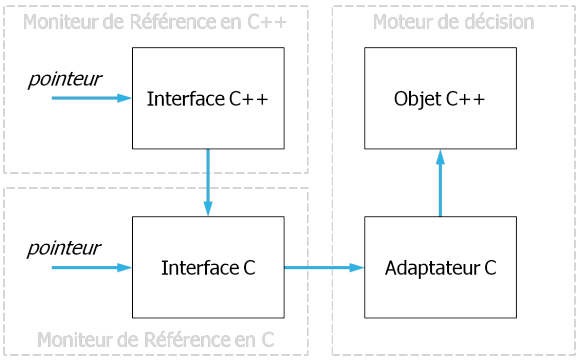



FIGURE 5.6 – Interfaces entre le moniteur de référence et un objet du moteur de décision

Ainsi, nous avons dû mettre en œuvre un patron de conception de type adaptateur pour mettre à disposition du moniteur de référence des pointeurs sur les

objets internes de notre moteur de décision (cf. figure 5.6). Pour que le moniteur puisse appeler des méthodes sur ces objets, nous définissons deux structures servant d'interfaces C/C++ et dont les membres sont des pointeurs sur les fonctions qui peuvent être appelées. Il est important que les offset et l'ordre des membres de ces structures soient rigoureusement identiques pour qu'ils puissent être utilisé invariablement par du code C et du code C++. Du coté du moteur de décisions, nous avons définis des wrappers en C dont le rôle est de rediriger les appels du moniteur de référence vers la véritable méthode de l'objet. Les listings 5.1 et 5.2 donnent un exemple de cela, tiré de notre prototype.

Cette conception permet une grande portabilité du moteur de décisions. Notamment, nous avons pu le réutiliser tel quel pour OpenJDK, JVMTI, Dalvik et Avian. Cela nous a ainsi permis de tester plusieurs stratégies d'instrumentation sans changer la logique de contrôle.

### 5.2.2 Implémentation du contrôle d'accès

Nous avons implémenté trois logiques de contrôle d'accès : du *multi-niveaux* (MLS/MCS), du *Domain & Type Enforcement* (DTE) et notre logique de contrôle par automate.

Le MLS/MCS a l'avantage d'être rapide et facile à implémenter. Malgré les défauts inhérents à cette logique, nous avions pour idée de l'utiliser dans le but de filtrer les élévations grossières de privilèges. En effet, le temps de décision de la logique DTE est supérieur à celui de MLS/MCS car le premier s'appuie sur de la comparaison de chaînes de caractères (les types) alors que le second ne fait intervenir que de la manipulation de masques de bits (1 bit par niveau/catégorie). Grâce à cela, nous espérions gagner en performance mais nous n'avons pu évaluer cette idée en pratique au regard des faibles performances de notre moniteur de référence.

La logique DTE consiste à prendre le type de sécurité des entités *Sujet* et *Objet* puis de rechercher dans un fichier texte toutes les règles qui correspondent ; la dernière étape consistant alors à comparer la permission demandée avec la permission requise. La difficulté est que cette étape de recherche pèse lourd sur les temps de décision surtout lorsqu'il s'agit d'utiliser des expressions régulières (cf. suggestion 4.10). Par ailleurs, notre modèle de contrôle impose de réaliser quatre comparaisons de chaînes par itération (c-à-d les types du *Sujet*, de l'*Objet* et la signature des membres) voire six si l'on utilise le contexte de sécurité étendu (cf. suggestion 4.3).

Or, les bases de données sont justement conçues pour indexer et rechercher des données dont la clef est une chaîne de caractères. C'est pourquoi nous nous

```
/* Forward declaration of interfaces */
struct c_interface_t;
struct cpp_interface_t;

/* Mapping to correct interface */
#ifdef __cplusplus
typedef struct cpp_interface_t interface_t;
#else
typedef struct c_interface_t interface_t;
#endif

/* C interface definition */
struct sejava_c_interface_t {

  /* Chaque fonction de l'interface retourne un code d'erreur.
   * La macro SEJAVA_CALL spécifie la convention d'appel utilisé.
   * Pour els systèmes 32 bits, nous imposons le CDECL pour forcer
   * l'appellant à nettoyer la pile d'exécution.
   */
  sejava_error_t (SEJAVA_CALL * check_invoke) (sejava_t * sejava,
    security_context_t * s_context, security_identifier_t * s_member,
    security_context_t * t_context, security_identifier_t * t_member);
};

/* C++ interface definition */
struct sejava_cpp_interface_t {

  /* Les premiers membres de l'interface C++ doivent être ceux
   * de l'interface C. En effet, celle ci pourrait être utilisé
   * par du code C non compilé en C++. Même si l'objet réel est
   * différent, les membres de l'interface C auront exatement
   * les mêmes offset dans ces deux structures. En pratique cela
   * reviens à émuler un héritage entre les deux interfaces.
   */
  struct sejava_c_interface_t functions;

#ifdef __cplusplus

  /* Ici, nous redefinissons les membres de l'interface C au
   * standard C++. Ceux-ci ne doivent pas être virtuels car cela
   * activerait la résolution dynamique de type (RTTI) et forcerait
   * le compilateur à insérer un membre d'identification de type
   * en entête de cette structure. Pour réduire l'overhead de
   * l'interface, nous pouvons inliner chaque méthode.
   */
  inline sejava_error_t check_execute(
    security_context_t * s_context, security_identifier_t * s_member,
    security_context_t * t_context, security_identifier_t * t_member)
  {
    /* On vérifie l'état de l'interface C */
    if (NULL == this->functions.check_invoke)
      return SEJAVA_ERROR_NOT_IMPLEMENTED;

    /* On appelle la méthode éponyme dans l'interface C */
    return this->functions.check_invoke(this,
      source_context, source_member,
      target_context, target_member);
  };
#endif
};

/* Il s'agit maintenant d'exporter les méthodes pour instancier/supprimer
 * l'objet interne qui implémente ces interfaces.
 */
extern SEJAVA_API sejava_error_t SEJava_Initialize(sejava_t ** sejava);
extern SEJAVA_API sejava_error_t SEJava_Release(sejava_t * sejava);
```

Listing 5.1 – Interfaces C/C++ exportées

```
#include "sejava-api.h"

/* ------------------------------------------------------------------- */
/* L'idée est d'instancier un objet C++ qui implémente toute la logique
 * de contrôle. Cet objet est ensuite exporté pour être utilisé par le
 * moniteur de référence. Sachant que le moniteur peut être développé
 * dans un langage autre que le C++, nous allons masquerons le type SEJava
 * en celui de l'interface C++.
 */
class SEJava : public sejava_t {
public:

  /* Il s'agit de créer ici les wrappers nécéssaires pour que les appels
     * sur l'interface C soient redirigés sur les méthodes de l'objet.
   * Pour cela, nous utilisons des méthodes statiques dont l'addresse
   * servira à initialiser les membres éponymes de l'interface C.
   */
static sejava_error_t SEJAVA_CALL check_invoke(sejava_t * sejava,
  security_context_t * s_context, security_identifier_t * s_member,
  security_context_t * t_context, security_identifier_t * t_member) {
    if (NULL == sejava) return SEJAVA_ERROR_INVALID_ENVIRONMENT;

    /* On appelle la méthode éponyme sur l'objet de type SEJava */
    return ((SEJava*)sejava)->check_invoke(
      source_context, source_member,
      target_context, target_member);
  };                                                          \

  /* Maintenant que les appels sont redirigés sur les méthodes de l'objet,
   * il ne nous reste plus qu'à définir l'implémentation de ces méthodes.
   * Celles-ci ne peuvent pas être virtuelles pour éviter le RTTI.
   */
  sejava_error_t check_invoke(
  security_context_t * s_context, security_identifier_t * s_member,
  security_context_t * t_context, security_identifier_t * t_member) {

    // [...]
  };

  /* Enfin, nous définissons un constructeur C++ pour cet objet
   * dont le but est d'initialiser tous les membres de son interface
   */
  SEJava(void) {

    /* On initianlise les membres de l'interface C */
    this->functions.check_invoke = SEJava::check_invoke;
  };
};

/* L'ultime étape consiste à exporter l'objet C++ ci-avant. La façon
 * la plus élégante est encore d'utiliser des méthodes d'allocation
 * et de déallocation.
 */
SEJAVA_API sejava_error_t SEJava_Initialize(sejava_t ** sejava) {
  /* On instancie l'objet demandé et c'est fini */
  if (NULL != sejava) (*sejava) = new SEJava();
  return (sejava ? SEJAVA_ERROR_NONE : SEJAVA_ERROR_NULL_POINTER);
}
SEJAVA_API sejava_error_t SEJava_Release(sejava_t * sejava) {
  /* On supprimer l'objet passé en paramètre */
  if (NULL != sejava) delete ((SEJava*)sejava);
  return (sejava ? SEJAVA_ERROR_NONE : SEJAVA_ERROR_NULL_POINTER);
}
```

Listing 5.2 – Implémentation dans le moteur de décision

sommes appuyés sur une base SQLite [8] pour stocker la politique DTE lors du fonctionnement de la JVM. En parallèle, nous avons cherché un moyen pour pré-calculer les privilèges de chaque objet et ainsi gagner nettement en performance. L'idée était d'insérer dans le binaire d'une classe Java ses types de sécurité et ses privilèges. Pour cela, les spécifications de la Machine Virtuelle Java [77] précisent qu'un compilateur peut ajouter librement à la fin d'un fichier `.class` un attribut supplémentaire non défini dans les spécifications. Pour y accéder, il suffit de mettre à jour les parsers de la JVM ou bien d'utiliser l'événement JVMTI `CLASS_FILE_LOAD_HOOK` pour accéder au binaire d'une classe chargée. Néanmoins, il n'est pas possible de pré-calculer la politique DTE de toutes les classes Java comme par exemple celles des applets Java. Pour cette raison nous avons donc conservé les deux systèmes : les tables SQL et l'attribut DTE supplémentaire dans le `.class`.

Les listings de la figure 5.7 donnent les fichiers de politique que nous avons utilisés et qui étaient chargés dans la base SQLite. Les fichiers `.vmc` et `.vmr`, pour *Virtual Machine Context* et *Virtual Machine Rules*, donnent respectivement les règles de labelisation et les règles de contrôle à appliquer. Les règles de contrôle définies dans le `.vmr` correspondent à l'application de notre suggestion 4.2 sur l'élimination du "bruit". L'idée étant de s'appuyer uniquement sur notre logique de contrôle par automates bien qu'il soit possible d'ajouter manuellement des règles de contrôle explicites.

Dans le chapitre 4 nous avions présenté une logique de contrôle basée sur des automates que nous avons notamment utilisée pour transcrire une politique JAAS en politique DTE. Pour construire de tels automates nous devons définir les règles de contrôle correspondantes à chacune des capacités JAAS. Ce travail, nous l'avons réalisé qu'une seule fois à partir des spécifications JAAS [20] et dont le résultat est donné en annexe A.

En pratique les automates mis en œuvre dans la traduction de politiques JAAS en SEJava sont certes nombreux mais petits. En effet, ceux-ci ne sont composés que d'un état initial contenant les règles DTE correspondantes aux capacités JAAS traduites et d'un état final qui soit accepte la relation si le domaine possède la capacité JAAS, soit la rejette dans le cas contraire. Sachant que n'importe quelle transition conduit à une acceptation ou un rejet, nous avons simplifié l'implémentation en enrichissant la politique DTE générale avec les règles des états initiaux ; celles-ci étant exprimées en fonction du type de sécurité du *domaine de protection* JAAS où elles ont été extraites. Bien entendu, cette optimisation ne permet pas de gérer les automates plus complexes comme ceux présentés en début de chapitre 4. Toutefois, cela s'est avéré suffisant dans le cadre de nos tests.

8. `https://www.sqlite.org`

```
1
2 #
3 # Hardcoded SEJava's type
4 #
5
6 Ljava/lang/Object;    object_j
7
8 Z            boolean_j
9 B            byte_j
10 C           char_j
11 S           short_j
12 I           int_j
13 J           long_j
14 F           float_j
15 D           double_j
16 V           void_j
17 E           enum_j
18
19 Ljava/lang/Boolean;   boolean_j
20 Ljava/lang/Byte;    byte_j
21 Ljava/lang/Character; char_j
22 Ljava/lang/Short;   short_j
23 Ljava/lang/Integer;   int_j
24 Ljava/lang/Long;    long_j
25 Ljava/lang/Float;   float_j
26 Ljava/lang/Double;    double_j
27 Ljava/lang/Void;    void_j
28
29 Ljava/lang/Class;   class_j
30 Ljava/lang/Number;    number_j
31 Ljava/lang/String;    string_j
32
33 #
34 # Massively used Java type
35 #
36
37 Ljava/lang/Throwable; exception_j
38 Ljava/lang/Thread;    thread_j
39 Ljava/lang/Process;   process_j
40 Ljava/lang/Runtime;   runtime_j
41 L*ClassLoader;      classloader_j
```

(a) Fichier `.vmc` définissant les règles de labelisation

```
1 allow *_j --{ * invoke * }--> *_j
2 allow *_j --{ * write * }--> *_j
3 allow *_j --{ * read * }--> *_j
```

(b) Fichier `.vmr` définissant les règles de contrôle

FIGURE 5.7 – Fichiers utilisés pour définir la politique DTE

### 5.2.3 Moteur d'apprentissage

Définir une politique DTE n'est pas aisée et notre première approche a été de suivre le principe de l'outil *audittoallow* de SELinux. C'est à dire, lancer l'application Java sans règle de contrôle et générer une règle de contrôle de type *allow* pour chaque relation observée et interdite. Cette approche fonctionne correctement et permet d'obtenir rapidement une politique de sécurité quoique peu pratique puisqu'elle correspond à un comportement observé qui n'est pas nécessairement légitime. Mais c'était surtout le fait de devoir exécuter une première fois une application dont on ne savait rien pour en déduire ce qu'elle avait le droit ou non de faire. Typiquement, cela ne fonctionnait pas lorsque la dite application était un malware Java.

Grâce à notre méthode pour passer d'une politique JAAS à une politique DTE, cette problématique n'avait plus lieu d'être. Par contre l'élément clef de cette traduction est le fait que l'on connaisse les spécifications de chaque capacité JAAS. De fait, si le développeur Java ne fournit pas cette information, la traduction est impossible. Or, lorsque l'on a commencé à utiliser l'injection de bytecodes, nous avons rapidement constaté que nous instrumentions des méthodes Java déjà instrumentées par JAAS. Notre idée était alors de développer un *SecurityManager* pour intercepter les appels à JAAS et déterminer quelles capacités JAAS permettent d'accéder à quelles méthodes de l'application. En somme, essayer de retrouver les spécifications des capacités JAAS simplement en observant les contrôles de sécurité de ce dernier. Néanmoins nous n'avons pas pu évaluer cette seconde approche par manque de temps.

Au final, l'implémentation du moteur de décision n'est pas aussi difficile que celle du moniteur de référence. En effet, on peut rapidement obtenir un composant fonctionnel en utilisant des briques logicielles sur étagère comme SQLite. Par contre, une implémentation inspirée de FLASK [106] permettrait d'obtenir de meilleures performances moyennant un effort d'ingénierie conséquent.

## 5.3 Résultats expérimentaux

Notre protocole de test repose principalement sur l'utilisation de malware Java. Nous avons utilisé ceux proposés par Metasploit [9] mais d'autres sont disponibles dans les bases d'exploits publiques [10]. L'idée est de se placer dans le cadre d'un site web infecté qu'un internaute lambda visiterait. Le navigateur ouvre ce site internet, identifie une applet Java dans la page HTML et lance la machine virtuelle de l'internaute pour l'exécuter (cf. figure 5.8).

9. `http://www.metasploit.com`
10. `http://www.1337day.com`

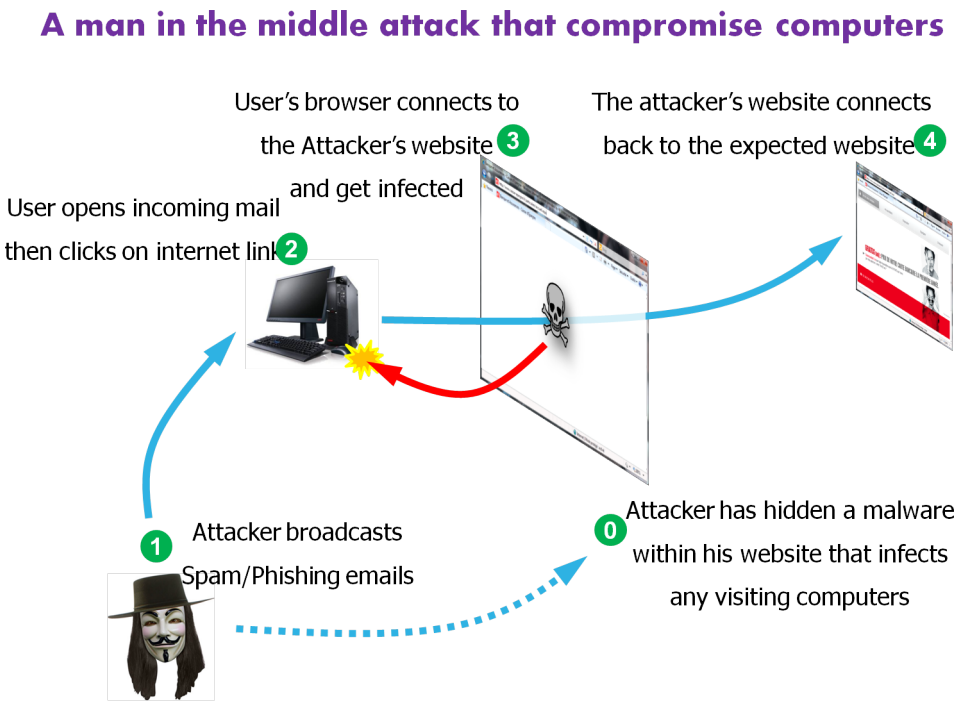



FIGURE 5.8 – Représentation de notre scénario d'expérimentation

### 5.3.1 Résultats d'expérimentation sur la CVE-2013-2460

Nous détaillons ici les résultats obtenus avec notre approche dans le cas d'un malware Java quelconque. En l'occurrence, il s'agit d'un exploit proposé par Metasploit pour la CVE-2013-2460 [11]. Nous donnons en section suivante un vision plus globale en effectuant la même expérimentation mais avec plusieurs Malware.

Comme nous l'avons plusieurs fois expliqué dans ce document, un malware Java se présente généralement sous la forme d'une applet malicieuse. Au moment où le classloader charge les classes de l'applet, JAAS détermine le *domaine de protection* le plus approprié pour chacune d'entre elles. C'est à dire soit un domaine JAAS équivalent à celui d'une application Java standard (peu de restrictions), soit un domaine aux privilèges limités (sandbox Java). Il est intéressant de noter que dans la plupart des situations, la machine virtuelle Java se contente de trois domaines JAAS : un domaine *système* qui regroupe toutes les classes noyau de la machine virtuelle Java ; un domaine *application* pour les applications Java standard ; et un domaine *applet* pour le sandboxing des applets.

Ainsi, lorsque la JVM exécute le malware relatif à la CVE-2013-2460, nous pouvons constater que celui-ci a pour but de désactiver JAAS afin d'instancier

11. `https://github.com/rapid7/metasploit-framework/tree/master/external/source/exploits/cve-2013-2460`

```
1  public class Exploit extends Applet
2  {
3    InvocationHandler invoc = null;
4    MethodHandles.Lookup look;
5
6    public Exploit()
7    {
8      try
9      {
10       ByteArrayOutputStream localByteArrayOutputStream = new
             ByteArrayOutputStream();
11       byte[] arrayOfByte = new byte[8192];
12
13       InputStream localInputStream = getClass().getResourceAsStream("
             DisableSecurityManagerAction.class");
14       int i;
15       while ((i = localInputStream.read(arrayOfByte)) > 0) {
16         localByteArrayOutputStream.write(arrayOfByte, 0, i);
17       }
18       arrayOfByte = localByteArrayOutputStream.toByteArray();
19
20       ProviderFactory localProviderFactory = ProviderFactory.
             getDefaultFactory();
21       Provider localProvider = localProviderFactory.createProvider(
             ExpProvider.class);
22       this.invoc = Proxy.getInvocationHandler(localProvider);
23       MethodHandles localMethodHandles = MethodHandles.class;
24
25       Method localMethod = localMethodHandles.getMethod("lookup", new Class
             [0]);
26       this.look = ((MethodHandles.Lookup)this.invoc.invoke(null, localMethod,
              new Object[0]));
27
28       Class localClass1 = loadClassUnderPrivContext("sun.org.mozilla.
             javascript.internal.Context");
29       Class localClass2 = loadClassUnderPrivContext("sun.org.mozilla.
             javascript.internal.DefiningClassLoader");
30       Class localClass3 = loadClassUnderPrivContext("sun.org.mozilla.
             javascript.internal.GeneratedClassLoader");
31
32       MethodHandle localMethodHandle1 = getMethod(localClass1, "enter",
             localClass1, new Class[0], true);
33
34       Class[] arrayOfClass = new Class[1];
35       arrayOfClass[0] = ClassLoader.class;
36
37       MethodHandle localMethodHandle2 = getMethod(localClass1, "
             createClassLoader", localClass3, arrayOfClass, false);
38
39       arrayOfClass = new Class[2];
40       arrayOfClass[0] = Class.forName("java.lang.String");
41       arrayOfClass[1] = new byte[0].getClass();
42
43       MethodHandle localMethodHandle3 = getMethod(localClass2, "defineClass",
              Class.class, arrayOfClass, false);
44
45       Object localObject1 = localMethodHandle1.invoke();
46       Object localObject2 = localMethodHandle2.invoke(localObject1, null);
47       Class localClass4 = localMethodHandle3.invoke(localObject2, "
             DisableSecurityManagerAction", arrayOfByte);
48
49       localClass4.newInstance();
50       Payload.main(null);
51     }
52     catch (Throwable localThrowable)
53     {
54     }
55   }
56   /* [...] */
57 }
```

FIGURE 5.9 – Code source du malware Metasploit pour la CVE-2013-2460, obtenu par rétro-ingénierie via JD-GUI

un chargeur de classes privilégiées et, par ce biais, de s'évader de la sandbox Java. En pratique, cela se traduit à terme par un appel au constructeur de la classe *Classloader* depuis la classe *Exploit* comme l'atteste le code source de ce malware en figure 5.9 (cf. ligne 49). Cela peut alors se traduire par la relation suivante :

$$\begin{aligned} & Contexte(loc_1) \xrightarrow{Signature(o_1::f_1)\ invoke\ Signature(o_3::f_1)} Contexte(loc_3) \\ \equiv & Signature(o_1) \xrightarrow{Signature(o_1::f_1)\ invoke\ Signature(o_3::f_1)} Signature(o_3) \\ \equiv & \text{"}Exploit\text{"} \xrightarrow{\text{"<init>" invoke "<init>"}} \text{"}Classloader\text{"} \end{aligned}$$

Avec $loc_1 \equiv Localisation(o_1)$ et $loc_3 \equiv Localisation(o_3)$

Quand notre protection est active, nous calculons dynamiquement la signature des classes chargées en mémoire avant même leur utilisation au titre de notre suggestion 4.3. Plus particulièrement, nous traduisons également les privilèges JAAS de ces classes à chaque fois qu'un nouveau *domaine de protection* est créé. Bien évidemment, sachant qu'un domaine JAAS regroupe plusieurs objets, nous mettons en cache les politiques DTE automatiquement générées. Ainsi, lorsque la JVM charge en mémoire la classe principale du malware - c'est à dire la classe *Exploit* - nous calculons immédiatement son étiquette DTE et déduisons dynamiquement ses privilèges à partir de son domaine de protection ; c'est à dire le domaine *applet* que JAAS vient de créer pour cette applet.

Dans ce but, nous interrogeons donc la JVM pour accéder au *domaine de protection* de la classe *Exploit* via un appel à la méthode JAAS *GetProtectionDomain* ; Le tableau 5.2 donne le résultat de cet appel. Ce domaine JAAS nous indique que la classe est chargée par le classloader *Applet2ClassLoader* et la politique de labelisation que nous utilisons indique que cela correspond à l'étiquette *applet_d* (cf. lignes 41 à 58 du listing A.1 en annexe A). La classe *Exploit* n'ayant pas d'étiquette prédéfinie, celle-ci hérite par défaut de l'étiquette de la classe Objet, soit *object_j* au titre de notre suggestion 4.8. Il en résulte que ce malware est automatiquement étiqueté *applet_d/object_j* juste après son chargement en mémoire.

Par ailleurs, le domaine JAAS *applet* nous est inconnu puisque JAAS vient juste de le créer afin de charger l'applet ; nous ne savons pas pour l'instant si celle-ci est malicieuse ou non. Ainsi, nous construisons un automate de contrôle qui reflète les capacités JAAS données au domaine *applet* (cf. figure 5.10.a) à partir duquel nous calculons dynamiquement une politique de contrôle DTE applicable (cf. figure 5.10.b) et dont nous pouvons anticiper les conséquences sur l'applet vis-à-vis des objets du système (cf. figure 5.10.c). Notamment, nous pouvons observer que la politique DTE interdit tout accès aux objets de type *Classloader*. Bien entendu, nous ne détaillons pas la politique complète pour plus de lisibilité.

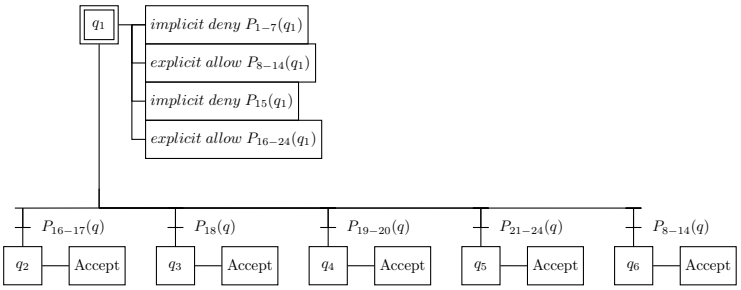
| Clée | Valeur |
|---|---|
| Classloader | sun.plugin2.applet.Applet2ClassLoader |
| Codesource | http ://192.168.0.135/ZomIzI2OT/ |
| Codesigners | n/a |
| Principals | n/a |

(a) Contexte de sécurité JAAS

| Permission | Cible | Actions |
|---|---|---|
| PropertyPermission | browser | read |
| PropertyPermission | browser.vendor | read |
| PropertyPermission | browser.version | read |
| PropertyPermission | file.separator | read |
| PropertyPermission | http.agent | read |
| PropertyPermission | java.class.version | read |
| PropertyPermission | java.specification.name | read |
| PropertyPermission | java.specification.vendor | read |
| PropertyPermission | java.specification.version | read |
| PropertyPermission | java.vendor | read |
| PropertyPermission | java.vendor.url | read |
| PropertyPermission | java.version | read |
| PropertyPermission | java.vm.name | read |
| PropertyPermission | java.vm.specification.name | read |
| PropertyPermission | java.vm.specification.vendor | read |
| PropertyPermission | java.vm.specification.version | read |
| PropertyPermission | java.vm.vendor | read |
| PropertyPermission | java.vm.version | read |
| PropertyPermission | javapi.* | read,write |
| PropertyPermission | javaplugin.version | read |
| PropertyPermission | javaplugin.vm.options | read |
| PropertyPermission | javaws.* | read,write |
| PropertyPermission | jnlp.* | read,write |
| PropertyPermission | line.separator | read |
| PropertyPermission | mrj.version | read |
| PropertyPermission | os.arch | read |
| PropertyPermission | os.name | read |
| PropertyPermission | os.version | read |
| PropertyPermission | path.separator | read |
| SecureCookiePermission | origin.file :// :-1 | |
| SocketPermission | localhost :1024 | listen,resolve |
| RuntimePermission | accessClassInPackage.sun.audio | |
| RuntimePermission | stopThread | |
| FilePermission | http ://192.168.0.135/ZomIzI2OT/- | read |

(b) Privilèges JAAS

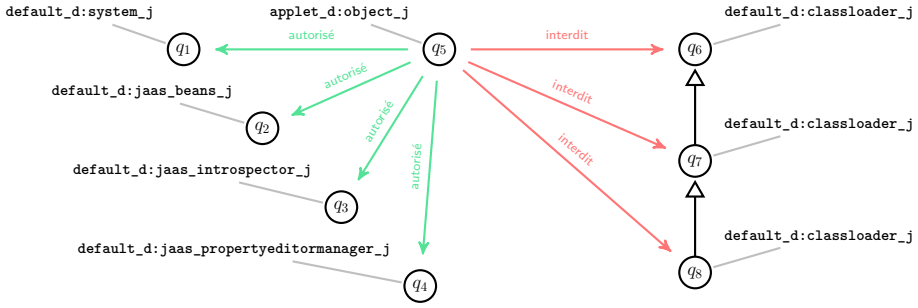
TABLE 5.2 – Domaine de protection du malware Metasploit pour la CVE-2013-2460




(a) Automate partiel calculé à partir des capacités JAAS du domaine *applet*

```
# Labelling rules
# (default is "object_j")
Ljava/lang/System;                          system_j
L*ClassLoader;                              classloader_j
Ljava/beans/Beans;                          jaas_beans_j
Ljava/beans/Introspector;                   jaas_introspector_j
Ljava/beans/PropertyEditorManager;          jaas_propertyeditormanager_j

jaas.classloader("Applet2ClassLoader")      applet_d

# Control rules
# ==== default rule to ignore JVM's noise ====
allow * ----> *

# ==== JAAS capacity (PropertyPermission, *, read) ====
allow applet_d:*_j --{ * invoke setDesignTime(Z)V }--> *_d:jaas_beans_j
allow applet_d:*_j --{ * invoke setGuiAvailable(Z)V }--> *_d:jaas_beans_j
allow applet_d:*_j --{ * invoke setBeanInfoSearchPath(String{[]})V }--> *_d:
      jaas_introspector_j
allow applet_d:*_j --{ * invoke registerEditor(Class,Class)V }--> *_d:
      jaas_propertyeditormanager_j
allow applet_d:*_j --{ * invoke setEditorSearchPath(String[])V }--> *_d:
      jaas_propertyeditormanager_j
allow applet_d:*_j --{ * invoke getProperties()Properties }--> *_d:system_j
allow applet_d:*_j --{ * invoke setProperties(Properties)V }--> *_d:system_j
allow applet_d:*_j --{ * invoke getProperty(String)String }--> *_d:system_j
allow applet_d:*_j --{ * invoke getProperty(String,String)String }--> *_d:system_j

# ==== JAAS capacity (RuntimePermission, createClassLoader, *) ====
deny applet_d:*_j --{ * invoke <init>()V }--> *_d:classloader_j
deny applet_d:*_j --{ * invoke <init>(*)V }--> *_d:classloader_j
```

(b) Politique DTE partielle en sortie de l'automate (a)




(c) Conséquence de (b) sur le système

FIGURE 5.10 – Transcription des privilèges du malware Metasploit en politique DTE

Au niveau de l'implémentation, nous appliquons ici une petite optimisation qui consiste à créer un ensemble de capacités JAAS connues et à y soustraire toutes les capacités données au domaine de protection pour lequel nous sommes en train de calculer la politique. Nous obtenons ainsi deux ensembles : celui des capacités JAAS données au domaine *applet* et un ensemble de capacités manquantes à ce domaine qui donnent respectivement lieu à un ensemble de règles de type *allow* et un autre de type *deny*. La complexité de gestion d'un véritable automate étant alors masqué par une simple soustraction entre deux ensembles (cf. figure 5.11).

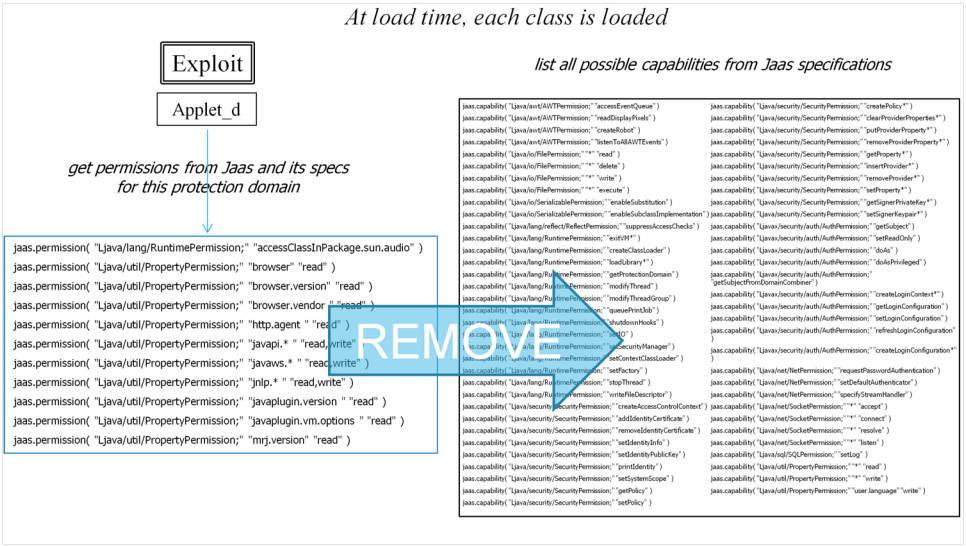



FIGURE 5.11 – Déduction des permissions JAAS manquantes au domaine de protection *applet*

Nous rappelons qu'à cet instant, le malware n'est toujours pas exécuté, seules ses classes ont été chargées par la JVM. Or nous connaissons maintenant les règles de contrôle DTE à appliquer aux classes du domaine *applet*. Nous pouvons donc laisser la JVM exécuter l'applet. Notamment, la JVM commence par créer une instance de la classe *Exploit* ce qui se traduit par un appel à sa méthode *<init>*. Cette dernière conduit à l'exploitation de la vulnérabilité CVE-2013-2460 et à la désactivation de JAAS via un appel privilégié à la méthode *SetSecurityManager* de l'API Java.

$$
\begin{aligned}
& Contexte_{Java}(loc_1) \xrightarrow{Signature(o_1::f_1)\ invoke\ Signature(o_2::f_1)} Contexte_{Java}(loc_2) \\
\equiv & \acute{E}tiquette_{DTE}(o_1) \xrightarrow{Signature(o_1::f_1)\ invoke\ Signature(o_2::f_1)} \acute{E}tiquette_{DTE}(o_2) \\
\equiv & \text{"}applet\_d/object\_j\text{"} \xrightarrow{\text{"}<init>\text{"}\ invoke\ \text{"}SetSecurityManager\text{"}} \text{"}system\_d/system\_j\text{"}
\end{aligned}
$$

Avec $loc_1 \equiv Localisation(o_1)$ et $loc_2 \equiv Localisation(o_2)$

Nous aurions pu agir à cet instant, mais une erreur de notre part dans la traduction de la capacité JAAS *setSecurityManager* a fait que nous n'avons pas

contrôlé correctement cet appel. Cependant, comme mentionné précédemment, le constructeur de la classe *Exploit* continue son exécution pour instancier un objet de type *Classloader*. Cela se traduit par un appel à la méthode <*init*> de la classe *ClassLoader* depuis la méthode <*init*> de la classe *Exploit* :

$$\begin{aligned} & Contexte_{Java}(loc_1) \xrightarrow{Signature(o_1::f_1)\ invoke\ Signature(o_3::f_1)} Contexte_{Java}(loc_3) \\ \equiv & \acute{E}tiquette_{DTE}(o_1) \xrightarrow{Signature(o_1::f_1)\ invoke\ Signature(o_3::f_1)} \acute{E}tiquette_{DTE}(o_3) \\ \equiv & \text{"}applet\_d/object\_j\text{"} \xrightarrow{\text{"<init>" } invoke \text{ "<init>"}} \text{"}system\_d/classloader\_j\text{"} \end{aligned}$$

Avec $loc_1 \equiv Localisation(o_1)$ et $loc_3 \equiv Localisation(o_3)$

Or l'une des règles DTE calculée pour le domaine *applet* interdit l'accès au constructeur de la classe *Classloader* (cf. lignes 27 et 28 de la figure 5.10.b). En effet, lorsque l'on compare la relation observée avec cette règle, nous voyons clairement qu'il y a correspondance entre les deux. Parce que la règle exprimée est de type *deny*, nous levons une exception pour bloquer l'appel. Le malware ne peut alors plus continuer son exécution et est donc contré.

*règle de contrôle : deny* $\text{"}applet\_d/*\_j\text{"} \xrightarrow{\text{"*" } invoke \text{ "<init>"}} \text{"}*\_d/classloader\_j\text{"}$

*relation observée :* $\text{"}applet\_d/object\_j\text{"} \xrightarrow{\text{"<init>" } invoke \text{ "<init>"}} \text{"}system\_d/classloader\_j\text{"}$

Au final, nous ne savons pas si l'applet fournie par Metasploit est malicieuse ou non. En effet, il pourrait simplement s'agir d'un bogue dans l'applet ou bien d'un oubli dans la politique JAAS. Par contre, nous avons pu observer que celle-ci tente une opération privilégiée normalement interdite par le domaine *applet*. C'est pourquoi nous l'avons bloquée afin de garantir les objectifs de sécurité de la sandbox Java. Le fait que nous ayons fait une erreur dans nos règles de traduction des capacités JAAS montre que même si nous échouons à empêcher une élévation de privilèges, nous restons en mesure d'empêcher les suivantes.

### 5.3.2 Résultats d'expérimentation sur une période d'une année

Nous avons conduit une série d'expérimentations dont le but était de vérifier l'efficacité de notre approche face à de multiples malwares Java. Pour cela, nous avons utilisé un système sous Linux CentOS 6.4 i686 et OpenJDK 1.7 update 4 build 20 dans leurs configurations par défaut. Le choix du navigateur importe peu car celui-ci délègue automatiquement à la JVM l'exécution de l'applet qui est ici malicieuse. Les malwares utilisés sont donnés par le tableau 5.4 qui couvre une période d'une année, de juin 2012 à juillet 2013.

| Vulnérabilité | Description de la vulnérabilité dans Metasploit |
|---|---|
| CVE-2012-1723 | Field Bytecode Verifier Cache Remote Code Execution |
| CVE-2012-4681 | Java 7 Applet Remote Code Execution |
| CVE-2012-5076 | AverageRangeStatisticImpl Remote Code Execution |
| CVE-2012-5076 | JAX-WS Remote Code Execution |
| CVE-2012-5088 | Method Handle Remote Code Execution |
| CVE-2013-1488 | Driver Manager Privileged toString() Remote Code Execution |
| CVE-2013-0422 | JMX Remote Code Execution |
| CVE-2013-2423 | Reflection Type Confusion Remote Code Execution |
| CVE-2013-0431 | Java JMX Remote Code Execution |
| CVE-2013-1493 | Java CMM Remote Code Execution |
| CVE-2013-2460 | ProviderSkeleton Insecure Invoke Method |

TABLE 5.4 – Malwares Metasploit utilisés

Dans un premier temps, nous avons testé chacun de ces malwares sans notre solution. Puis, dans un second temps, nous avons relancé le même test mais en activant notre protection. En cas de défaillance de notre solution le malware réussit à prendre le contrôle de notre machine de test. Si notre approche s'avère efficace et empêche le malware de sortir de la sandbox Java alors un message d'erreur sous Metasploit apparaît indiquant l'échec de l'attaque. Le tableau 5.5 présente les résultats obtenus pour chacun des malwares du tableau 5.4.

| Vulnérabilité | État de la protection | |
|---|---|---|
| | désactivée (permissive) | activée (enforce) |
| CVE-2012-1723 | exploitation réussie | Crash de l'application |
| CVE-2012-4681 | exploitation réussie | exploitation échouée |
| CVE-2012-5076 | exploitation réussie | exploitation échouée |
| CVE-2012-5076 | exploitation réussie | exploitation échouée |
| CVE-2012-5088 | exploitation réussie | exploitation échouée |
| CVE-2013-1488 | exploitation réussie | exploitation échouée |
| CVE-2013-0422 | exploitation réussie | exploitation échouée |
| CVE-2013-2423 | exploitation réussie | exploitation échouée |
| CVE-2013-0431 | exploitation réussie | exploitation échouée |
| CVE-2013-1493 | exploitation échouée | exploitation échouée |
| CVE-2013-2460 | exploitation réussie | exploitation échouée |

TABLE 5.5 – Résultat des tentatives d'exploitation avec et sans la protection

Ainsi, on constate que lorsque la protection est inactive, l'attaque réussit à chaque fois. La seule exception concerne la CVE-2012-1493 dont l'exploit ne fonctionne que sous Windows XP ou Windows 7 alors que notre machine de test est sous CentOS. Par contre, lorsque l'on active la protection, on constate qu'aucun malware n'est en mesure de compromettre la JVM alors que la vulnérabilité exploitée est bien présente comme le confirme le premier test. Le cas de la CVE-2012-1723 entraîne un crash de la machine virtuelle Java car notre moyen de bloquer une interaction est de lever une exception qui ne semble pas être gérée par la JVM dans ce cas précis.

Au final, cette expérimentation montre les bénéfices concrets de notre approche

en empêchant un objet malicieux de contourner les restrictions de JAAS. Les seules règles que nous avons définies pour contrer 1 an de malware Java consistent à traduire simplement les capacités JAAS en règles DTE ; l'application de ces capacités étant alors garantie par notre logique de contrôle par automates. Or sans ces automates, il nous aurait fallu définir des dizaines de milliers de règles DTE pour obtenir un résultat similaire.

Mais cette expérimentation montre surtout que la garantie du principe de sandboxing Java est quasi immédiate. En effet, les objectifs de sécurité d'une sandbox Java sont plutôt simples et donc sa garantie également. Il serait ainsi intéressant d'améliorer les performances de ce prototype de contrôle pour adresser des objectifs de sécurité plus complexes comme l'isolation de deux applications OSGi.

## 5.4 Discussions

Nous avons organisé ce chapitre en deux axes principaux avec d'une part l'implémentation de notre prototype et de l'autre comment ce dernier ce comporte face à de véritables malware Java.

Le développement d'un moniteur de référence pour la JVM s'est avéré être la partie la plus difficile car les relations que nous pouvons contrôler dépendent directement des possibilités de dialogue avec la JVM. Ainsi nous sommes partis sur l'idée de modifier une JVM Open Source mais face aux efforts d'ingénierie à fournir, nous avons préféré utiliser une interface d'instrumentation standard : JVMTI. Cependant, JVMTI ne nous a pas donné entière satisfaction du fait que cette interface n'a pas été conçue pour intégrer un module de sécurité au sein de la JVM. C'est pourquoi nous avons exploré l'idée de l'injection de bytecodes Java pour détourner les appels de méthode que nous voulions surveiller.

Nous avons pu évaluer autant de méthodes d'instrumentation par le fait que le moteur de décisions était indépendant du moniteur de référence. Par ce biais, nous avons pu mettre en commun les composants de notre moteur de décisions dont la logique de contrôle par automate, le moteur DTE, le moteur MLS/MCS qui ne nous a pas vraiment servi au final et un embryon de moteur d'apprentissage de règles de contrôle.

Ainsi, notre prototype de recherche nous a permis d'observer ce qu'il se passe à l'intérieur de la JVM lorsque celle-ci exécute un malware. Pour cela, nous avons utilisé l'outil Metasploit qui est disponible gratuitement afin de lancer des attaques contre OpenJDK. Il en ressort que nous sommes effectivement en mesure de bloquer tous les malware Java sortis sur une période d'une année en traduisant dynamiquement les politiques JAAS en politique DTE.

Chapitre 6

# Conclusion et perspectives

*La preuve par l'analogie est une fraude.*

Bjarne Stroustrup [107]

Cette thèse a été réalisée dans un contexte industriel, au sein des laboratoires Bell-Labs d'Alcatel-Lucent. Le besoin de traiter les problématiques de sécurité de Java est parti du constat que plus de la moitié des produits Alcatel-Lucent utilisent une Machine Virtuelle Java. En effet, le moindre problème de sécurité lié à Java, impacte directement ces produits aussi bien d'un point de vue économique qu'en terme d'images auprès des clients de l'entreprise.

Ainsi, notre état de l'art a montré qu'il n'existait pas de mécanisme de protection satisfaisant pour Java. Notamment le chapitre 2 débute par un historique des vulnérabilités Java afin d'en dégager un modèle d'attaque aussi générique que possible. Nous avons alors pu dégager quatre types d'attaques récurrentes sur Java : la corruption de mémoire, la confusion de types, les défauts de contrôle d'accès et l'abus de l'inspection de pile. À partir de cette analyse, nous avons compris que le modèle de sécurité standard de Java était insuffisant pour traiter ces quatre cas. Plus particulièrement, ni les mécanismes de sécurité liés au langage, ni ceux de la JVM, ni l'approche de contrôle d'accès JAAS n'arrivent à satisfaire les objectifs de sécurité de la JVM et notamment son principe de sandboxing des applications. Les recherches scientifiques sur le sujet font également ce constat, mais aucune d'entre elles n'apporte une solution satisfaisante. En effet, l'analyse statique est réputée comme faible par la littérature [30]. La coloration dynamique et la preuve formelle peuvent difficilement être mises en œuvre du fait de la complexité algorithmique intrinsèque de ces approches. Et

enfin, les approches par contrôle d'accès obligatoire se heurtent à la difficulté de définir une politique de sécurité. Dès lors, il apparaît évident que Java nécessite un véritable moniteur de référence à même de garantir l'application de politiques de sécurité.

C'est pourquoi, nous avons abordé le problème en proposant une modélisation générique des systèmes à objets. Le chapitre 3 part ainsi sur l'idée que si nous sommes en mesure d'observer les relations entre objets du système alors nous saurons les contrôler et, in fine, garantir l'application de politiques. Nous nous sommes donc intéressés à modéliser les grandes notions des systèmes à objets que sont l'objet, le membre, la localisation et la signature. De même nous avons identifié trois types de relations élémentaires exprimables à l'aide de ces notions de base. Il s'agit de la référence, de l'interaction qui met en œuvre les membres des objets et les trois grands types de flux (information, données et activité). Il s'avère que cette formalisation est suffisante pour modéliser les systèmes à objets particuliers moyennant l'ajout de quelques notions spécifiques comme la classe, le prototype et l'interface. Ainsi notre modèle couvre les systèmes à classes, à prototypes, à objets répartis et tout autres systèmes orientés objet.

De cette modélisation, nous faisons apparaître la possibilité d'exprimer les relations entre objets à l'aide d'automates. Plus particulièrement, nous avons montré dans le chapitre 4 que l'utilisation d'automates peut se substituer à la logique JAAS dans le but de mieux gérer les élévations de privilèges. Notamment, lorsque l'on utilise les objectifs de sécurité de JAAS, nous sommes capables de traiter les quatre cas du modèle d'attaques établi dans l'état de l'art alors même que JAAS se révèle inefficace. De ce constat, nous avons donc utilisé ces automates pour réécrire les privilèges JAAS associés aux objets de la JVM en règles de contrôle au format DTE. Il en ressort que cette retranscription des politiques JAAS en politique DTE peut se faire de façon totalement dynamique. C'est à dire qu'au moment où JAAS crée un *domaine de protection* nous utilisons nos automates pour déterminer dynamiquement toutes les relations que les objets de ce *domaine de protection* sont en droit de réaliser ou non.

Néanmoins, cette approche de contrôle par automate nécessite un véritable moniteur de référence pour être utilisable en pratique. En effet, les trois relations élémentaires que nous avons définies sont contrôlables seulement si nous sommes en mesure de les observer. Ainsi, dans le chapitre 5 nous présentons les différentes stratégies d'instrumentation de JVM, classées de la plus difficile à mettre en œuvre à la plus facile. Nous avons donc testé la possibilité de modifier une machine virtuelle Java existante afin de l'adapter à nos besoins. Nous avons aussi regardé du côté de l'interface d'instrumentation standard JVMTI pour développer un module de sécurité compatible avec la plupart des JVM. Et enfin, nous avons évalué la possibilité d'injecter des bytecodes dans le code des classes Java dans le but de détourner les seuls appels de méthodes pertinents à contrôler. Ces trois stratégies d'instrumentations ont leurs propres avantages et inconvénients qui influent sur les possibilités de dialogue avec la JVM et donc sur le contrôle. Ce point-ci est, de notre expérience, le plus difficile d'un point de vue technique.

Nous avons présenté en fin de chapitre les résultats obtenus avec notre logique de contrôle par automates. Notamment, il apparaît que nous sommes en mesure de contrer tous les malware Java sur une période d'une année.

Au final, le modèle de contrôle basé sur les relations observables entre objets couplé à l'idée d'utiliser des automates est clairement une avancée par rapport aux travaux de recherche antérieurs. Notamment, cela permet dynamiquement de traduire des objectifs de sécurité définis statiquement en relations à contrôler. Le contrôle qui est ainsi réalisé par chaque automate est extrêmement dynamique puisque celui-ci se fait à l'échelle de chaque objet du système où les politiques à appliquer sont automatiquement recalculées à chaque fois que les besoins de sécurité de l'application évoluent. La clef de ce résultat scientifique réside en deux points :

— D'une part en travaillant directement sur les objectifs de sécurité, nous concentrons nos efforts de contrôle uniquement sur ce qui est pertinent à contrôler. Cela élimine donc naturellement le bruit des politiques favorisant ainsi la ré-utilisabilité, la facilité de maintenance, etc. En bref, toutes les qualités que l'on attend d'une bonne politique de sécurité.

— D'autre part, le travail que réalise ces automates consiste essentiellement à rendre explicites tous les sous-entendus des objectifs de sécurité. C'est à dire à respecter le principe de *tout ce qui n'est pas explicitement autorisé est implicitement interdit* mais aussi celui qui est de dire que *tout ce qui n'est pas implicitement interdit est implicitement autorisé*. Ces deux principes réunis permettent donc de générer automatiquement les règles de contrôle qui reflètent avec exactitude les objectifs de sécurité du système. Cet aspect là résout à lui seul la limitation historique des approches obligatoires.

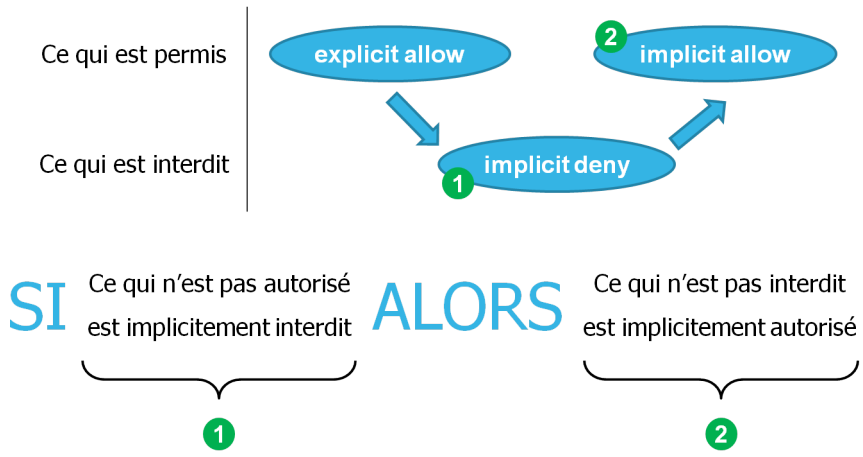



FIGURE 6.1 – Résumé du principe de fonctionnement des automates de contrôle

L'avancée de la thèse se mesure aussi à l'étendue des perspectives des résultats. Nous allons citer les principales pistes de recherche qui découlent de nos propositions.

Les systèmes embarqués peuvent également tirer parti de notre travail. En effet, nous avons progressé en réduisant le surcoût du contrôle ce qui permet d'envisager d'utiliser nos méthodes dans des systèmes à faibles ressources. Ainsi, les cartes à puces sont des cibles privilégiées qui peuvent utiliser directement nos solutions étant donnée que l'instrumentation est plus aisée voir immédiate via des moniteurs de référence embarqués dans la noyau du système d'exploitation de la carte. Ensuite, les objets connectés tels que des composants mobiles de sécurité peuvent utiliser nos approches de contrôle d'accès et donc venir compléter les mécanismes de protection matériels et logiciels existants en exploitant le côté entièrement dynamique de notre solution. On peut notamment imaginer des jetons matériels hétérogènes (Hardware Security Modules) qui communiquent entre eux en échangeant les objectifs de sécurité pour garantir la protection de le vie privée et des données personnelles partagées. Des premiers résultats ont été obtenus dans le cadre des systèmes embarqués Android bien que ceux-ci n'aient pas été développés dans ce document.

L'informatique en nuages repose sur des systèmes dynamiques (cycle de vie des systèmes virtualisées, des conteneurs et des applications) qui nécessitent une grande adaptabilité et flexibilité de la part des mécanismes de sécurité. Ainsi, il semble possible de protéger différentes zones de confiance dans un Cloud en déléguant des capacités qui peuvent être combinées localement sous la forme d'automates pour garantir des propriétés de sécurité avancées et réparties. Ces idées ont notamment été présentées lors d'une conférence invitée à l'atelier sur la sécurité du Cloud [108].

Les systèmes communicants personnels nécessitent la confiance des utilisateurs. Pour cela, il faut redonner la maîtrise de leurs données aux utilisateurs. Dans ce cadre, notre approche par automate est tout à fait adaptée puisque l'utilisateur peut non seulement piloter ces automates via des langages naturels mais aussi les visualiser et interagir avec eux pour autoriser ou empêcher dynamiquement les accès et les partages des données. Là aussi des premiers résultats ont été obtenus dans le cadre d'Android qui n'ont pas été détaillés ici.

Enfin, il est possible d'utiliser nos automates pour piloter d'autres éléments que des politiques de sécurité. Il semble en effet envisageable de garantir aussi des objectifs fonctionnels via notre approche. Nous pourrions, par exemple, créer dynamiquement des contraintes d'intégrité au sein d'une base de données en lieu et place des traditionnelles contraintes SQL qui se révèlent trop statiques dans le contexte du "Big Data". De même, nous pourrions imaginer porter notre approche par automate à des domaines autres que la cyber-sécurité. En effet, les processus d'entreprises ont souvent des contraintes contextualisées comme par exemple un trader qui n'a pas le droit de dépasser son seuil d'investissement autorisé lorsqu'il s'agit de flux financier.

Annexe A

# Règles de traduction des politique JAAS en SEJava

Les deux listings ci-dessous contiennent les règles qui permettent d'exprimer les privilèges donnés à un domaine de protection en un ensemble de règles de contrôle au format *Domain & Type Enforcement*. Les entrées de type `jaas.permission` servent à ajouter manuellement des capacités JAAS à un type de sécurité. Les capacités JAAS que nous avons ajoutées sont celles par défaut et n'ont ici qu'un caractère informatif.

```
1
2  #
3  # Well-known JAAS protection domain
4  #
5
6  protection domain {
7
8      # Default jvm permission as defined into java.policy
9      jaas.permission( "Ljava/lang/RuntimePermission;"    "stopThread" )
10
11     jaas.permission( "Ljava/net/SocketPermission;"      "localhost:1024-"
                "listen" )
12
13     jaas.permission( "Ljava/util/PropertyPermission;"   "java.version"
                "read" )
14     jaas.permission( "Ljava/util/PropertyPermission;"   "java.vendor"
              "read" )
15     jaas.permission( "Ljava/util/PropertyPermission;"   "java.vendor.url"
                "read" )
16     jaas.permission( "Ljava/util/PropertyPermission;"   "java.class.version"
                  "read" )
17     jaas.permission( "Ljava/util/PropertyPermission;"   "os.name"
            "read" )
18     jaas.permission( "Ljava/util/PropertyPermission;"   "os.version"
              "read" )
19     jaas.permission( "Ljava/util/PropertyPermission;"   "os.arch"
            "read" )
20     jaas.permission( "Ljava/util/PropertyPermission;"   "file.separator"
                "read" )
```

```
21      jaas.permission( "Ljava/util/PropertyPermission;"   "path.separator"
                "read" )
22      jaas.permission( "Ljava/util/PropertyPermission;"   "line.separator"
                "read" )
23
24      jaas.permission( "Ljava/util/PropertyPermission;"   "java.specification.
            version"    "read" )
25      jaas.permission( "Ljava/util/PropertyPermission;"   "java.specification.
            vendor"     "read" )
26      jaas.permission( "Ljava/util/PropertyPermission;"   "java.specification.
            name"     "read" )
27
28      jaas.permission( "Ljava/util/PropertyPermission;"   "java.vm.
            specification.version"    "read" )
29      jaas.permission( "Ljava/util/PropertyPermission;"   "java.vm.
            specification.vendor"     "read" )
30      jaas.permission( "Ljava/util/PropertyPermission;"   "java.vm.
            specification.name"     "read" )
31      jaas.permission( "Ljava/util/PropertyPermission;"   "java.vm.version"
                "read" )
32      jaas.permission( "Ljava/util/PropertyPermission;"   "java.vm.vendor"
                "read" )
33      jaas.permission( "Ljava/util/PropertyPermission;"   "java.vm.name"
                "read" )
34
35      # Required java permission to dump security policy
36      jaas.permission( "Ljava/lang/RuntimePermission;"    "getProtectionDomain"
             )
37      jaas.permission( "Ljava/security/SecurityPermission;" "getPolicy" )
38
39    } default_d
40
41  protection domain {
42      jaas.classloader( "Lsun/plugin2/applet/Applet2ClassLoader;" )
43      jaas.classloader( "Lsun/applet/AppletClassLoader;" )
44
45      # Default applet permission as hardcoded into class sun.plugin2.applet.
            Applet2ClassLoader
46      jaas.permission( "Ljava/lang/RuntimePermission;"    "accessClassInPackage
            .sun.audio" )
47      jaas.permission( "Ljava/util/PropertyPermission;"   "browser"
              "read" )
48      jaas.permission( "Ljava/util/PropertyPermission;"   "browser.version"
                "read" )
49      jaas.permission( "Ljava/util/PropertyPermission;"   "browser.vendor"
                "read" )
50      jaas.permission( "Ljava/util/PropertyPermission;"   "http.agent"
                "read" )
51      jaas.permission( "Ljava/util/PropertyPermission;"   "javapi.*"
                "read,write" )
52      jaas.permission( "Ljava/util/PropertyPermission;"   "javaws.*"
                "read,write" )
53      jaas.permission( "Ljava/util/PropertyPermission;"   "jnlp.*"
              "read,write" )
54      jaas.permission( "Ljava/util/PropertyPermission;"   "javaplugin.version"
                  "read" )
55      jaas.permission( "Ljava/util/PropertyPermission;"   "javaplugin.vm.
            options"       "read" )
56      jaas.permission( "Ljava/util/PropertyPermission;"   "mrj.version"
                "read" )
57
58    } applet_d
59
60  protection domain {
61      jaas.classloader( "Lsun/misc/Launcher$AppClassLoader;" )
62
63    } application_d
64
65  protection domain {
66      jaas.classloader( "<bootstrap-classloader>" )
67
68      jaas.codesource.classpath( "file:*/jre/lib/rt.jar" )
69      jaas.codesource.classpath( "file:*/jre/lib/jce.jar" )
70      jaas.codesource.classpath( "file:*/jre/lib/jsse.jar" )
```

```
71
72   } system_d
73
74 #
75 # Reserved JAAS security label
76 #
77
78 Ljava/awt/Toolkit;                jaas_toolkit_j
79 Ljava/awt/Graphics2d;             jaas_graphics2d_j
80 Ljava/awt/Robot;                 jaas_robot_j
81 Ljava/awt/Window;                jaas_window_j
82
83 Ljava/io/File;                   jaas_file_j
84 Ljava/io/FileInputStream;            jaas_fileinputstream_j
85 Ljava/io/FileOutputStream;             jaas_fileoutputstream_j
86 Ljava/io/RandomAccessFile;             jaas_randomaccess_j
87 Ljava/util/zip/ZipFile;              jaas_zipfile_j
88
89 Ljava/io/ObjectInputStream;             jaas_objectinputstream_j
90 Ljava/io/ObjectOutputStream;            jaas_objectoutputstream_j
91
92 Ljava/lang/Runtime;               jaas_runtime_j
93 Ljava/lang/System;                jaas_system_j
94 Ljava/lang/Class;                 jaas_class_j
95 Ljava/lang/ClassLoader;              jaas_classloader_j
96 Ljava/lang/Thread;                jaas_thread_j
97 Ljava/lang/ThreadGroup;              jaas_threadgroup_j
98 Ljava/net/Socket;                 jaas_socket_j
99 Ljava/net/URL;                    jaas_url_j
100 Ljava/net/URLConnection;              jaas_urlconnection_j
101 Ljava/net/HttpURLConnection;            jaas_httpurlconnection_j
102 Ljava/rmi/activation/ActivationGroup;      jaas_activationgroup_j
103 Ljava/rmi/server/RMISocketFactory;         jaas_rmisocketfactory_j
104 Ljava/net/URLClassLoader;            jaas_urlclassloader_j
105 Ljava/security/SecureClassLoader;        jaas_secureclassloader_j
106
107 Ljava/net/Authenticator;             jaas_authenticator_j
108
109 Ljava/security/AccessControlContext;       jaas_accesscontrolcontext_j
110 Ljava/security/Identity;              jaas_identity_j
111 Ljava/security/IdentityScope;           jaas_identityscope_j
112 Ljava/security/Policy;                jaas_policy_j
113 Ljava/security/Provider;              jaas_provider_j
114 Ljava/security/Security;              jaas_security_j
115 Ljava/security/Signer;                jaas_signer_j
116
117 Ljavax/security/auth/Subject;           jaas_subject_j
118 Ljavax/security/auth/SubjectDomainCombiner;   jaas_subjectdomaincombiner_j
119 Ljavax/security/auth/login/LoginContext;    jaas_logincontext_j
120 Ljavax/security/auth/login/Configuration;   jaas_configuration_j
121
122 Ljava/net/MulticastSocket;             jaas_multicastsocket_j
123 Ljava/net/DatagramSocket;             jaas_datagramsocket_j
124 Ljava/net/ServerSocket;               jaas_serversocket_j
125 Ljava/net/InetAddress;                jaas_inetaddress_j
126
127 Ljava/sql/DriverManager;              jaas_drivermanager_j
128
129 Ljava/beans/Beans;                 jaas_beans_j
130 Ljava/beans/Introspector;             jaas_introspector_j
131 Ljava/beans/PropertyEditorManager;         jaas_propertyeditormanager_j
132 Ljava/util/Locale;                 jaas_locale_j
```

Listing A.1 – Règles de labelisation des domaines de protection JAAS

```
1
2 #
3 # This file was created from http://docs.oracle.com/javase/7/docs/technotes/
      guides/security/permissions.html
4 #
5
6 #
```

```
7  # ===== java.awt.AWTPermission =====
8  #
9
10 jaas.capability( "Ljava/awt/AWTPermission;" "accessEventQueue" ) {
11   Ljava/awt/Toolkit;                getSystemEventQueue()Ljava/awt/EventQueue
         ;
12 }
13
14 jaas.capability( "Ljava/awt/AWTPermission;" "readDisplayPixels" ) {
15   Ljava/awt/Graphics2d;             setComposite(Ljava/awt/Composite;)V
16 }
17
18 jaas.capability( "Ljava/awt/AWTPermission;" "createRobot" ) {
19   Ljava/awt/Robot;                <init>()V
20   Ljava/awt/Robot;                <init>(Ljava/awt/GraphicsDevice;)V
21 }
22
23 jaas.capability( "Ljava/awt/AWTPermission;" "listenToAllAWTEvents" ) {
24   Ljava/awt/Toolkit;                addAWTEventListener(Ljava/awt/event/
         AWTEventListener;J)V
25   Ljava/awt/Toolkit;                removeAWTEventListener(Ljava/awt/event/
         AWTEventListener;)V
26 }
27
28 # If java.awt.AWTPermission "showWindowWithoutWarningBanner" is set,
29 # the window will be displayed without a banner warning that the window
30 # was created by an applet. It it's not set, such a banner will be displayed.
31 #
32 #jaas.capability( "Ljava/awt/AWTPermission;" "showWindowWithoutWarningBanner"
       ) {
33 # Ljava/awt/Window;               <init>()V
34 #}
35
36 #
37 # ===== java.io.FilePermission =====
38 #
39
40 jaas.capability( "Ljava/io/FilePermission;" "*" "read" ) {
41   Ljava/io/FileInputStream;           <init>(Ljava/lang/String;)V
42   Ljava/io/FileInputStream;           <init>(Ljava/io/File;)V
43
44   Ljava/io/File;                  exists()Z
45   Ljava/io/File;                  canRead()Z
46   Ljava/io/File;                  isFile()Z
47   Ljava/io/File;                  isDirectory()Z
48   Ljava/io/File;                  isHidden()Z
49   Ljava/io/File;                  lastModified()J
50   Ljava/io/File;                  length()J
51   Ljava/io/File;                  list(*)[Ljava/lang/String;
52   Ljava/io/File;                  listFiles(*)[Ljava/io/File;
53
54   #Ljava/io/RandomAccessFile;          <init>(Ljava/lang/String;Ljava/lang/
         String;)Z
55   #Ljava/io/RandomAccessFile;          <init>(Ljava/io/File;Ljava/lang/
         String;)Z
56   # (where mode is "r" in both of these)
57
58   Ljava/util/zip/ZipFile;             <init>(Ljava/lang/String;)V
59 }
60
61 jaas.capability( "Ljava/io/FilePermission;" "*" "delete" ) {
62   Ljava/io/File;                  delete()Z
63   Ljava/io/File;                  deleteOnExit()V
64 }
65
66 jaas.capability( "Ljava/io/FilePermission;" "*" "write" ) {
67   Ljava/io/FileOutputStream;          <init>(Ljava/io/File;)V
68   Ljava/io/FileOutputStream;          <init>(Ljava/lang/String;)V
69   Ljava/io/FileOutputStream;          <init>(Ljava/lang/String;Z)V
70
71   #Ljava/io/RandomAccessFile;          <init>(Ljava/lang/String;Ljava/lang/
         String;)Z
72   #Ljava/io/RandomAccessFile;          <init>(Ljava/io/File;Ljava/lang/
         String;)Z
```

```
73    # (where mode is "w" in both of these)
74
75    Ljava/io/File;                  canWrite()Z
76    Ljava/io/File;                  createNewFile()Z
77    Ljava/io/File;                  createTempFile(Ljava/lang/String;Ljava/lang
          /String;)Ljava/io/File;
78    Ljava/io/File;                  createTempFile(Ljava/lang/String;Ljava/lang
          /String;Ljava/io/File;)Ljava/io/File;
79    Ljava/io/File;                  mkdir()Z
80    Ljava/io/File;                  mkdirs()Z
81    Ljava/io/File;                  renameTo(Ljava/io/File;)Z
82    Ljava/io/File;                  setLastModified(J)Z
83    Ljava/io/File;                  setReadOnly()Z
84  }
85
86  jaas.capability( "Ljava/io/FilePermission;" "*" "execute" ) {
87    Ljava/lang/Runtime;               exec(Ljava/lang/String;)Ljava/lang/
          Process;
88    Ljava/lang/Runtime;               exec(Ljava/lang/String;[Ljava/lang/String
          ;)Ljava/lang/Process;
89    Ljava/lang/Runtime;               exec([Ljava/lang/String;)Ljava/lang/
          Process;
90    Ljava/lang/Runtime;               exec([Ljava/lang/String;[Ljava/lang/
          String;)Ljava/lang/Process;
91  }
92
93  #
94  # ===== java.io.SerializablePermission =====
95  #
96
97  jaas.capability( "Ljava/io/SerializablePermission;" "enableSubstitution" ) {
98    Ljava/io/ObjectInputStream;           enableResolveObject(Z)Z
99
100   Ljava/io/ObjectOutputStream;          enableResolveObject(Z)Z
101 }
102
103 jaas.capability( "Ljava/io/SerializablePermission;" "
        enableSubclassImplementation" ) {
104   Ljava/io/ObjectInputStream;           <init>()Z
105
106   Ljava/io/ObjectOutputStream;          <init>()Z
107 }
108
109 #
110 # ===== java.lang.reflect.ReflectPermission =====
111 #
112
113 jaas.capability( "Ljava/lang/reflect/ReflectPermission;" "
        suppressAccessChecks" ) {
114   Ljava/lang/reflect/AccessibleObject;      setAccessible()V
115 }
116
117 #
118 # ===== java.lang.RuntimePermission =====
119 #
120
121 jaas.capability( "Ljava/lang/RuntimePermission;" "exitVM*" ) {
122   Ljava/lang/Runtime;               exit(I)V
123   Ljava/lang/Runtime;               runFinalizersOnExit(Z)V
124
125   Ljava/lang/System;                exit(I)V
126   Ljava/lang/System;                runFinalizersOnExit(Z)V
127 }
128
129 #jaas.capability( "Ljava/lang/RuntimePermission;" "getClassLoader" ) {
130 # Ljava/lang/Class;                forName(Ljava/lang/String;ZLjava/lang/
        ClassLoader;)Ljava/lang/Class;
131 # # (when loader is null, and the caller's class loader is not null)
132 #}
133
134 # If the caller's class loader is null, or is the same as or an ancestor of
        the class loader
135 # for the class whose class loader is being requested, no permission is
        needed. Otherwise,
```

```
136 # java.lang.RuntimePermission "getClassLoader" is required.
137 #
138 #jaas.capability( "Ljava/lang/RuntimePermission;" "getClassLoader" ) {
139 # Ljava/lang/Class;                getClassLoader()Ljava/lang/ClassLoader;
140 #
141 # Ljava/lang/ClassLoader;              getSystemClassLoader()Ljava/lang/
      ClassLoader;
142 # Ljava/lang/ClassLoader;              getParent()Ljava/lang/ClassLoader;
143 #
144 # Ljava/lang/Thread;                 getContextClassLoader()Ljava/lang/
      ClassLoader;
145 #}
146
147 # Default checkMemberAccess does not require any permissions if "this" class'
      s classloader is
148 # the same as that of the caller. Otherwise, it requires java.lang.
      RuntimePermission "accessDeclaredMembers".
149 # If this class is in a package, java.lang.RuntimePermission "
      accessClassInPackage.{pkgName}" is also required.
150 #
151 #jaas.capability( "Ljava/lang/RuntimePermission;" "accessDeclaredMembers*" )
      {
152 # Ljava/lang/Class;                getDeclaredClasses()[Ljava/lang/Class;
153 # Ljava/lang/Class;                getDeclaredFields()[Ljava/lang/reflect/
      Field;
154 # Ljava/lang/Class;                getDeclaredMethods()[Ljava/lang/reflect/
      Method;
155 # Ljava/lang/Class;                getDeclaredConstructors()[Ljava/lang/
      reflect/Constructor;
156 # Ljava/lang/Class;                getDeclaredField(Ljava/lang/String;)Ljava/
      lang/reflect/Field;
157 # Ljava/lang/Class;                getDeclaredMethod(*)Ljava/lang/reflect/
      Method;
158 # Ljava/lang/Class;                getDeclaredConstructor(*)Ljava/lang/reflect
      /Constructor;
159 #}
160
161 # Default checkMemberAccess does not require any permissions when the access
      type is Member.PUBLIC.
162 # If this class is in a package, java.lang.RuntimePermission "
      accessClassInPackage.{pkgName}" is required.
163 #
164 #jaas.capability( "Ljava/lang/RuntimePermission;" "accessClassInPackage*" ) {
165 # Ljava/lang/Class;                getClasses()[Ljava/lang/Class;
166 # Ljava/lang/Class;                getFields()[Ljava/lang/reflect/Field;
167 # Ljava/lang/Class;                getMethods()[Ljava/lang/reflect/Method;
168 # Ljava/lang/Class;                getConstructors()[Ljava/lang/reflect/
      Constructor;
169 # Ljava/lang/Class;                getField(Ljava/lang/String;)Ljava/lang/
      reflect/Field;
170 # Ljava/lang/Class;                getMethod(*)Ljava/lang/reflect/Method;
171 # Ljava/lang/Class;                getConstructor(*)Ljava/lang/reflect/
      Constructor;
172 #}
173
174 jaas.capability( "Ljava/lang/RuntimePermission;" "createClassLoader" ) {
175   Ljava/lang/ClassLoader;             <init>()V
176   Ljava/lang/ClassLoader;             <init>(Ljava/lang/ClassLoader;)V
177
178   Ljava/net/URLClassLoader;           <init>(*)V
179
180   Ljava/security/SecureClassLoader;       <init>(*)V
181 }
182
183 jaas.capability( "Ljava/lang/RuntimePermission;" "loadLibrary*" ) {
184   Ljava/lang/Runtime;               load(Ljava/lang/String;)V
185   Ljava/lang/Runtime;               loadLibrary(Ljava/lang/String;)V
186
187   Ljava/lang/System;                load(Ljava/lang/String;)V
188   Ljava/lang/System;                loadLibrary(Ljava/lang/String;)V
189 }
190
191 jaas.capability( "Ljava/lang/RuntimePermission;" "getProtectionDomain" ) {
```

```
192   Ljava/lang/Class;                getProtectionDomain()Ljava/security/
          ProtectionDomain;
193 }
194
195 jaas.capability( "Ljava/lang/RuntimePermission;" "modifyThread" ) {
196   Ljava/lang/Thread;                 checkAccess()V
197   Ljava/lang/Thread;                 interrupt()V
198   Ljava/lang/Thread;                 suspend()V
199   Ljava/lang/Thread;                 resume()V
200   Ljava/lang/Thread;                 setPriority(I)V
201   Ljava/lang/Thread;                 setName(Ljava/lang/String;)V
202   Ljava/lang/Thread;                 setDaemon(Z)V
203   Ljava/lang/Thread;                 stop()V
204   Ljava/lang/Thread;                 stop(Ljava/lang/Throwable;)V
205
206   Ljava/lang/ThreadGroup;              stop()V
207   Ljava/lang/ThreadGroup;              interrupt()V
208 }
209
210 jaas.capability( "Ljava/lang/RuntimePermission;" "modifyThreadGroup" ) {
211   Ljava/lang/Thread;                 <init>()V
212   Ljava/lang/Thread;                 <init>(Ljava/lang/Runnable;)V
213   Ljava/lang/Thread;                 <init>(Ljava/lang/String;)V
214   Ljava/lang/Thread;                 <init>(Ljava/lang/Runnable;Ljava/lang/
          String;)V
215   Ljava/lang/Thread;                 <init>(Ljava/lang/ThreadGroup;*)V
216   Ljava/lang/Thread;                 enumerate([Ljava/lang/Thread;)I
217
218   Ljava/lang/ThreadGroup;              <init>(Ljava/lang/String;)V
219   Ljava/lang/ThreadGroup;              <init>(Ljava/lang/ThreadGroup;Ljava/
          lang/String;)V
220   Ljava/lang/ThreadGroup;              checkAccess()V
221   Ljava/lang/ThreadGroup;              enumerate([Ljava/lang/Thread;)I
222   Ljava/lang/ThreadGroup;              enumerate([Ljava/lang/Thread;Z)I
223   Ljava/lang/ThreadGroup;              enumerate([Ljava/lang/ThreadGroup;)I
224   Ljava/lang/ThreadGroup;              enumerate([Ljava/lang/ThreadGroup;Z)I
225   Ljava/lang/ThreadGroup;              getParent()Ljava/lang/ThreadGroup;
226   Ljava/lang/ThreadGroup;              setDaemon(Z)V
227   Ljava/lang/ThreadGroup;              setMaxPriority(I)V
228   Ljava/lang/ThreadGroup;              suspend()V
229   Ljava/lang/ThreadGroup;              resume()V
230   Ljava/lang/ThreadGroup;              destroy()V
231   Ljava/lang/ThreadGroup;              stop()V
232   Ljava/lang/ThreadGroup;              interrupt()V
233 }
234
235 jaas.capability( "Ljava/lang/RuntimePermission;" "queuePrintJob" ) {
236   Ljava/awt/Toolkit;                 getPrintJob(Ljava/awt/Frame;Ljava/lang/
          String;Ljava/util/Properties;)Ljava/awt/PrintJob;
237 }
238
239 jaas.capability( "Ljava/lang/RuntimePermission;" "readFileDescriptor" ) {
240   Ljava/io/FileInputStream;            <init>(Ljava/io/FileDescriptor;)Z
241 }
242
243 jaas.capability( "Ljava/lang/RuntimePermission;" "shutdownHooks" ) {
244   Ljava/lang/Runtime;                addShutdownHook(Ljava/lang/Thread;)V
245   Ljava/lang/Runtime;                removeShutdownHook(Ljava/lang/Thread;)Z
246 }
247
248 jaas.capability( "Ljava/lang/RuntimePermission;" "setIO" ) {
249   Ljava/lang/System;                 setIn(Ljava/io/InputStream;)V
250   Ljava/lang/System;                 setOut(Ljava/io/PrintStream;)V
251   Ljava/lang/System;                 setErr(Ljava/io/PrintStream;)V
252 }
253
254 jaas.capability( "Ljava/lang/RuntimePermission;" "setSecurityManager" ) {
255   Ljava/lang/System;                 setSecurityManager(Ljava/lang/
          SecurityManager;)V
256 }
257
258 jaas.capability( "Ljava/lang/RuntimePermission;" "setContextClassLoader" ) {
259   Ljava/lang/Thread;                 setContextClassLoader(Ljava/lang/
          ClassLoader;)V
```

```
260 }
261
262 jaas.capability( "Ljava/lang/RuntimePermission;" "setFactory" ) {
263   Ljava/net/ServerSocket;              setSocketFactory(*)V
264
265   Ljava/net/Socket;                setSocketImplFactory(*)V
266
267   Ljava/net/URL;                   setURLStreamHandlerFactory(*)V
268
269   Ljava/net/URLConnection;             setContentHandlerFactory(*)V
270   Ljava/net/URLConnection;             setFileNameMap(Ljava/net/FileNameMap;)V
271
272   Ljava/net/HttpURLConnection;           setFollowRedirects(Z)V
273
274   Ljava/rmi/activation/ActivationGroup;      createGroup(*)Ljava/rmi/
          activation/ActivationGroup;
275   Ljava/rmi/activation/ActivationGroup;      setSystem(Ljava/rmi/activation/
          ActivationSystem;)V
276
277   Ljava/rmi/server/RMISocketFactory;         setSocketFactory(*)V
278 }
279
280 jaas.capability( "Ljava/lang/RuntimePermission;" "stopThread" ) {
281   #Ljava/lang/Thread;               stop()V
282   # If the current thread is trying to stop a thread other than itself.
283
284   #Ljava/lang/Thread;               stop(Ljava/lang/Throwable;)V
285   # If the current thread is trying to stop a thread other than itself or obj
          is not an instance of ThreadDeath.
286
287   Ljava/lang/ThreadGroup;              stop()V
288 }
289
290 jaas.capability( "Ljava/lang/RuntimePermission;" "writeFileDescriptor" ) {
291   Ljava/io/FileOutputStream;             <init>(Ljava/io/FileDescriptor;)Z
292 }
293
294 #
295 # ===== java.security.SecurityPermission =====
296 #
297
298 jaas.capability( "Ljava/security/SecurityPermission;" "
        createAccessControlContext" ) {
299   Ljava/security/AccessControlContext;      <init>(Ljava/security/
          AccessControlContext;Ljava/security/DomainCombiner;)V
300   Ljava/security/AccessControlContext;      getDomainCombiner()Ljava/security
          /DomainCombiner;
301 }
302
303 jaas.capability( "Ljava/security/SecurityPermission;" "addIdentityCertificate
        " ) {
304   Ljava/security/Identity;             addCertificate(*)V
305 }
306
307 jaas.capability( "Ljava/security/SecurityPermission;" "
        removeIdentityCertificate" ) {
308   Ljava/security/Identity;             removeCertificate(*)V
309 }
310
311 jaas.capability( "Ljava/security/SecurityPermission;" "setIdentityInfo" ) {
312   Ljava/security/Identity;             setInfo(Ljava/lang/String;)V
313 }
314
315 jaas.capability( "Ljava/security/SecurityPermission;" "setIdentityPublicKey"
        ) {
316   Ljava/security/Identity;             setPublicKey(Ljava/security/PublicKey;)
          V
317 }
318
319 jaas.capability( "Ljava/security/SecurityPermission;" "printIdentity" ) {
320   Ljava/security/Identity;             toString(*)Ljava/lang/String;
321 }
322
323 jaas.capability( "Ljava/security/SecurityPermission;" "setSystemScope" ) {
```

```
324   Ljava/security/IdentityScope;          setSystemScope()V
325 }
326
327 jaas.capability( "Ljava/security/SecurityPermission;" "getPolicy" ) {
328   Ljava/security/Policy;                 getPolicy()Ljava/security/Policy;
329 }
330
331 jaas.capability( "Ljava/security/SecurityPermission;" "setPolicy" ) {
332   Ljava/security/Policy;                 setPolicy(Ljava/security/Policy;)V
333 }
334
335 jaas.capability( "Ljava/security/SecurityPermission;" "createPolicy*" ) {
336   Ljava/security/Policy;                 getInstance(Ljava/lang/String;*)Ljava/
          security/Policy;
337 }
338
339 jaas.capability( "Ljava/security/SecurityPermission;" "
      clearProviderProperties*" ) {
340   Ljava/security/Provider;               clear()V
341 }
342
343 jaas.capability( "Ljava/security/SecurityPermission;" "putProviderProperty*"
      ) {
344   Ljava/security/Provider;               put(Ljava/lang/Object;Ljava/lang/Object
          ;)Ljava/lang/Object;
345 }
346
347 jaas.capability( "Ljava/security/SecurityPermission;" "removeProviderProperty
      *" ) {
348   Ljava/security/Provider;               remove(Ljava/lang/Object;)Ljava/lang/
          Object;
349 }
350
351 jaas.capability( "Ljava/security/SecurityPermission;" "getProperty*" ) {
352   Ljava/security/Security;               getProperty(Ljava/lang/String;)Ljava/
          lang/String;
353 }
354
355 jaas.capability( "Ljava/security/SecurityPermission;" "insertProvider*" ) {
356   Ljava/security/Security;               addProvider(Ljava/security/Provider;)I
357   Ljava/security/Security;               insertProviderAt(Ljava/security/
          Provider;I)I
358 }
359
360 jaas.capability( "Ljava/security/SecurityPermission;" "removeProvider*" ) {
361   Ljava/security/Security;               removeProvider(Ljava/lang/String;)V
362 }
363
364 jaas.capability( "Ljava/security/SecurityPermission;" "setProperty*" ) {
365   Ljava/security/Security;               setProperty(Ljava/lang/String;Ljava/
          lang/String;)V
366 }
367
368 jaas.capability( "Ljava/security/SecurityPermission;" "getSignerPrivateKey*"
      ) {
369   Ljava/security/Signer;                 getPrivateKey()Ljava/security/
          PrivateKey;
370 }
371
372 jaas.capability( "Ljava/security/SecurityPermission;" "setSignerKeypair*" ) {
373   Ljava/security/Signer;                 setKeyPair(Ljava/security/KeyPair;)V
374 }
375
376 #
377 # ===== javax.security.auth.AuthPermission =====
378 #
379
380 jaas.capability( "Ljavax/security/auth/AuthPermission;" "getSubject" ) {
381   Ljavax/security/auth/Subject;          getSubject(Ljava/security/
          AccessControlContext;)Ljavax/security/auth/Subject;
382 }
383
384 jaas.capability( "Ljavax/security/auth/AuthPermission;" "setReadOnly" ) {
385   Ljavax/security/auth/Subject;          setReadOnly()V
```

```
386 }
387
388 jaas.capability( "Ljavax/security/auth/AuthPermission;" "doAs" ) {
389   Ljavax/security/auth/Subject;          doAs(Ljavax/security/auth/Subject;
          Ljava/security/PrivilegedAction;)Ljava/lang/Object;
390   Ljavax/security/auth/Subject;          doAs(Ljavax/security/auth/Subject;
          Ljava/security/PrivilegedExceptionAction;)Ljava/lang/Object;
391 }
392
393 jaas.capability( "Ljavax/security/auth/AuthPermission;" "doAsPrivileged" ) {
394   Ljavax/security/auth/Subject;          doAsPrivileged(Ljavax/security/auth/
          Subject;Ljava/security/PrivilegedAction;Ljava/security/
          AccessControlContext;)Ljava/lang/Object;
395   Ljavax/security/auth/Subject;          doAsPrivileged(Ljavax/security/auth/
          Subject;Ljava/security/PrivilegedExceptionAction;Ljava/security/
          AccessControlContext;)Ljava/lang/Object;
396 }
397
398 jaas.capability( "Ljavax/security/auth/AuthPermission;" "
        getSubjectFromDomainCombiner" ) {
399   Ljavax/security/auth/SubjectDomainCombiner;    getSubject()Ljavax/security/
          auth/Subject;
400 }
401
402 jaas.capability( "Ljavax/security/auth/AuthPermission;" "createLoginContext*"
        ) {
403   Ljavax/security/auth/login/LoginContext;     <init>(Ljava/lang/String;)V
404   Ljavax/security/auth/login/LoginContext;     <init>(Ljava/lang/String;Ljavax
          /security/auth/Subject;)V
405   Ljavax/security/auth/login/LoginContext;     <init>(Ljava/lang/String;Ljavax
          /security/auth/callback/CallbackHandler;)V
406   Ljavax/security/auth/login/LoginContext;     <init>(Ljava/lang/String;Ljavax
          /security/auth/Subject;Ljavax/security/auth/callback/CallbackHandler;)
          V
407 }
408
409 jaas.capability( "Ljavax/security/auth/AuthPermission;" "
        getLoginConfiguration" ) {
410   Ljavax/security/auth/login/Configuration;    getConfiguration()Ljavax/
          security/auth/login/Configuration;
411 }
412
413 jaas.capability( "Ljavax/security/auth/AuthPermission;" "
        setLoginConfiguration" ) {
414   Ljavax/security/auth/login/Configuration;    setConfiguration(Ljavax/
          security/auth/login/Configuration;)V
415 }
416
417 jaas.capability( "Ljavax/security/auth/AuthPermission;" "
        refreshLoginConfiguration" ) {
418   Ljavax/security/auth/login/Configuration;    refresh(V)V
419 }
420
421 jaas.capability( "Ljavax/security/auth/AuthPermission;" "
        createLoginConfiguration*" ) {
422   Ljavax/security/auth/login/Configuration;    getInstance(Ljava/lang/String
          ;*)Ljavax/security/auth/login/Configuration;
423   Ljavax/security/auth/login/Configuration;    getInstance(Ljava/lang/String;*
          Ljava/lang/String;)Ljavax/security/auth/login/Configuration;
424   Ljavax/security/auth/login/Configuration;    getInstance(Ljava/lang/String;*
          Ljava/security/Provider;)Ljavax/security/auth/login/Configuration;
425 }
426
427 #
428 # ===== java.net.NetPermission =====
429 #
430
431 jaas.capability( "Ljava/net/NetPermission;" "requestPasswordAuthentication" )
        {
432   Ljava/net/Authenticator;             requestPasswordAuthentication(Ljava/net
          /InetAddress;ILjava/lang/String;Ljava/lang/String;Ljava/lang/String;)
          Ljava/net/PasswordAuthentication;
433 }
434
```

```
435 jaas.capability( "Ljava/net/NetPermission;" "setDefaultAuthenticator" ) {
436   Ljava/net/Authenticator;            setDefault(Ljava/net/Authenticator;)V
437 }
438
439 jaas.capability( "Ljava/net/NetPermission;" "specifyStreamHandler" ) {
440   Ljava/net/URL;                  <init>(*)V
441 }
442
443 #
444 # ===== java.net.SocketPermission =====
445 #
446
447 jaas.capability( "Ljava/net/SocketPermission;" "*" "accept" ) {
448   Ljava/net/MulticastSocket;            joinGroup(Ljava/net/InetAddress;)V
449   Ljava/net/MulticastSocket;            leaveGroup(Ljava/net/InetAddress;)V
450
451   Ljava/net/DatagramSocket;           send(Ljava/net/DatagramPacket;)V
452   Ljava/net/DatagramSocket;           receive(Ljava/net/DatagramPacket;)V
453
454   Ljava/net/ServerSocket;             accept()Ljava/net/ServerSocket;
455   Ljava/net/ServerSocket;             implAccept(Ljava/net/ServerSocket;)V
456 }
457
458 jaas.capability( "Ljava/net/SocketPermission;" "*" "connect" ) {
459   Ljava/net/MulticastSocket;            joinGroup(Ljava/net/InetAddress;)V
460   Ljava/net/MulticastSocket;            leaveGroup(Ljava/net/InetAddress;)V
461
462   Ljava/net/DatagramSocket;           send(Ljava/net/DatagramPacket;)V
463
464   Ljava/net/Socket;               <init>(*)V
465 }
466
467 jaas.capability( "Ljava/net/SocketPermission;" "*" "resolve" ) {
468   Ljava/net/DatagramSocket;           send(Ljava/net/DatagramPacket;)V
469   Ljava/net/DatagramSocket;           getLocalAddress()Ljava/net/InetAddress;
470
471   Ljava/net/InetAddress;              getHostName()Ljava/lang/String;
472   Ljava/net/InetAddress;              getAllByName(Ljava/lang/String;)[Ljava/
          net/InetAddress;
473   Ljava/net/InetAddress;              getLocalHost()Ljava/net/InetAddress;
474 }
475
476 jaas.capability( "Ljava/net/SocketPermission;" "*" "listen" ) {
477   Ljava/net/ServerSocket;             <init>(*)V
478
479   Ljava/net/DatagramSocket;           <init>(*)V
480
481   Ljava/net/MulticastSocket;            <init>(*)V
482 }
483
484 #
485 # ===== java.sql.SQLPermission =====
486 #
487
488 jaas.capability( "Ljava/sql/SQLPermission;" "setLog" ) {
489   Ljava/sql/DriverManager;            setLogWriter(Ljava/io/PrintWriter;)V
490   Ljava/sql/DriverManager;            setLogStream(Ljava/io/PrintWriter;)V
491 }
492
493 #
494 # ===== java.util.PropertyPermission =====
495 #
496
497 jaas.capability( "Ljava/util/PropertyPermission;" "*" "read" ) {
498   Ljava/beans/Beans;                setDesignTime(Z)V
499   Ljava/beans/Beans;                setGuiAvailable(Z)V
500
501   Ljava/beans/Introspector;             setBeanInfoSearchPath([Ljava/lang/
          String;)V
502
503   Ljava/beans/PropertyEditorManager;        registerEditor(Ljava/lang/Class;
          Ljava/lang/Class;)V
504   Ljava/beans/PropertyEditorManager;        setEditorSearchPath([Ljava/lang/
          String;)V
```

```
505
506   Ljava/lang/System;                getProperties()Ljava/util/Properties;
507   Ljava/lang/System;                setProperties(Ljava/util/Properties;)V
508   Ljava/lang/System;                getProperty(Ljava/lang/String;)Ljava/lang
          /String;
509   Ljava/lang/System;                getProperty(Ljava/lang/String;Ljava/lang/
          String;)Ljava/lang/String;
510 }
511
512 jaas.capability( "Ljava/util/PropertyPermission;" "*" "write" ) {
513   Ljava/beans/Beans;                setDesignTime(Z)V
514   Ljava/beans/Beans;                setGuiAvailable(Z)V
515
516   Ljava/beans/Introspector;           setBeanInfoSearchPath([Ljava/lang/
          String;)V
517
518   Ljava/beans/PropertyEditorManager;        registerEditor(Ljava/lang/Class;
          Ljava/lang/Class;)V
519   Ljava/beans/PropertyEditorManager;        setEditorSearchPath([Ljava/lang/
          String;)V
520
521   Ljava/lang/System;                getProperties()Ljava/util/Properties;
522   Ljava/lang/System;                setProperties(Ljava/util/Properties;)V
523   Ljava/lang/System;                setProperty(Ljava/lang/String;Ljava/lang/
          String;)Ljava/lang/String;
524 }
525
526 jaas.capability( "Ljava/util/PropertyPermission;" "user.language" "write" ) {
527   Ljava/util/Locale;                setDefault(Ljava/util/Locale$Category;
          Ljava/util/Locale;)V
528 }
```

Listing A.2 – Règles de traduction des capacités JAAS

Annexe B

# Implémentation du cas d'étude Java

Nous proposons ici une implémentation du cas d'étude présenté dans [83]. Les listings B.1, B.2, B.3, B.4, B.5, B.7 donnent le code source des classes java de ce cas d'étude. Les listings B.8 et B.9 sont des scripts windows qui permettent respectivement de compiler et lancer le cas d'étude. Le listing B.6 donne la politique Java à utiliser.

## Le maçon, la petite fille et le réfrigérateur

Les classes *Maçon* et *Fille* implémentent l'interface *Personne*. La classe *Réfrigérateur* a la responsabilité d'appeler JAAS pour vérifier que le code appelant est suffisamment privilégié pour y déposer ou prendre un objet.

```
1 public interface Person {
2   public Object request(Person person);
3
4   public Object take();
5 }
```

Listing B.1 – Code source de la classe Personne

```
1 public class Builder implements Person {
2
3   protected Fridge fridge;
4
```

```
5   public Builder(Fridge fridge) {
6     this.fridge = fridge;
7   }
8
9   public Object request(Person person) {
10    return person.take();
11  }
12
13  public Object take() {
14    return this.fridge.take();
15  }
16 }
```

Listing B.2 – Code source de la classe Maçon

```
1  import java.security.*;
2
3  public class Daughter implements Person {
4
5    protected Fridge fridge;
6
7    public Daughter(Fridge fridge) {
8      this.fridge = fridge;
9    }
10
11   public Object request(Person person) {
12     return person.take();
13   }
14
15   public Object take() {
16
17     // Check if current scenario disables JAAS stackinspection or not
18     if ((Usecase.scenario == 5) || (Usecase.scenario == 6)) {
19
20       // Access the fridge with elevated privileges
21       //   --> the callstack will be inspected until Daughter's frame
22       return AccessController.doPrivileged(new PrivilegedAction() {
23         public Object run() {
24           // privileged code goes here
25           return fridge.take();
26         }
27       });
28
29     } else {
30
31       // Access the fridge with no elevated privileges
32       //   --> the whole callstack will be inspected
33       return this.fridge.take();
34     }
35   }
36 }
```

Listing B.3 – Code source de la classe Fille

```
1  import java.security.*;
2
3
4  public class Fridge {
5
6    private java.util.Stack<Object> food = new java.util.Stack<Object>();
7
8    public Object take() {
9
10     SecurityManager jaas = System.getSecurityManager();
11     if (jaas != null) {
12
13       // Invoke JAAS
14       String item = this.food.peek().getClass().getName();
15       jaas.checkPermission(new FridgePermission(item, "take"));
16     }
```

```
17
18      // JAAS access check
19      return this.food.pop();
20    }
21
22    public Object put(Object article) {
23
24      SecurityManager jaas = System.getSecurityManager();
25      if (jaas != null) {
26
27        // Invoke JAAS
28        String item = article.getClass().getName();
29        jaas.checkPermission(new FridgePermission(item, "put"));
30      }
31
32      // JAAS access check
33      return this.food.push(article);
34    }
35  }
```

Listing B.4 – Code source de la classe Réfrigérateur

## Permission JAAS permettant de contrôler l'accès au réfrigérateur

La permission *FridgePermission* a le même fonctionnement que la permission JAAS *PropertyPermission* à cela près qu'elle prend en paramètre un nom d'objet et une action qui peut être *take* ou *put* pour respectivement prendre et déposer un objet dans le *Réfrigérateur*.

Le listing B.6 donne la politique Java à utiliser. Celle-ci autorise les objets de l'archive *trusted.jar* (*Fille* et *Réfrigérateur*) à prendre et déposer n'importe quel objet dans le *Réfrigérateur*. Les objets de l'archive *not-trusted.jar* (le *Maçon*) peuvent juste y déposer des objets.

```
1  import java.util.*;
2  import java.security.*;
3
4  public final class FridgePermission extends BasicPermission {
5
6    // Supported privileges
7    private final static int NONE = 0x0;
8    private final static int TAKE = 0x1;
9    private final static int PUT  = 0x2;
10     private final static int ALL  = TAKE|PUT;
11
12   private int bitmask = NONE;
13   private String actions = null;
14
15   public FridgePermission(String name, String actions) {
16     super(name, actions);
17
18     // Compute privileges
19     if (null != actions) {
20       for (String privilege : actions.split(",")) {
21         if (privilege.equalsIgnoreCase("take")) this.bitmask |= TAKE;
22         if (privilege.equalsIgnoreCase("put"))  this.bitmask |= PUT;
23       }
```

```
24      }
25    }
26
27    public PermissionCollection newPermissionCollection() {
28      return new FridgePermissionCollection();
29    }
30
31    public boolean implies(Permission p) {
32      // Check permission type
33      if (!(p instanceof FridgePermission)) return false;
34
35      // Check if that permission is a subset of this permission
36      FridgePermission that = (FridgePermission) p;
37      return ((this.bitmask & that.bitmask) == that.bitmask) && super.implies(
            that);
38    }
39
40    public boolean equals(Object obj) {
41      // Check if same object
42      if (obj == this) return true;
43
44      // Check permission type
45      if (! (obj instanceof FridgePermission)) return false;
46
47      // Check if that permission is a subset of this permission
48      FridgePermission that = (FridgePermission) obj;
49      return (this.bitmask == that.bitmask) && (this.getName().equals(that.
            getName()));
50    }
51
52    static String getActions(int bitmask) {
53
54      StringBuilder actions = new StringBuilder();
55      boolean is_empty = true;
56
57      if ((bitmask & TAKE) == TAKE) {
58        is_empty = false; actions.append("take");
59      }
60
61      if ((bitmask & PUT) == PUT) {
62        if (!is_empty) actions.append(",");
63        is_empty = false; actions.append("take");
64      }
65
66      return actions.toString();
67    }
68
69    int getBitmask() {
70      return this.bitmask;
71    }
72
73    public synchronized String getActions() {
74      if (null == this.actions)
75        this.actions = getActions(this.bitmask);
76
77      return this.actions;
78    }
79  }
80
81  final class FridgePermissionCollection extends PermissionCollection {
82
83    // Set of permission names
84    private transient Map<String,FridgePermission> permissions;
85
86    public FridgePermissionCollection() {
87      this.permissions = new HashMap<String,FridgePermission>(32);
88    }
89
90    public void add(Permission p) {
91
92      // Check permission type
93      if (! (p instanceof FridgePermission))
94        throw new IllegalArgumentException("invalid permission: " + p);
95
```

```
96      // Check write attribute
97      if (isReadOnly())
98        throw new SecurityException("attempt to add a Permission to a readonly
              PermissionCollection");
99
100     // Check if permission is already known or not
101     FridgePermission permission = (FridgePermission) p;
102     String target = permission.getName();
103
104     synchronized (this) {
105       FridgePermission found = (FridgePermission) permissions.get(target);
106       if (found != null) {
107         int old_bitmask = found.getBitmask();
108         int new_bitmask = permission.getBitmask();
109         if (old_bitmask != new_bitmask) {
110           String actions = FridgePermission.getActions(old_bitmask |
                  new_bitmask);
111           this.permissions.put(target, new FridgePermission(target, actions))
                  ;
112         }
113       } else {
114         this.permissions.put(target, permission);
115       }
116     }
117   }
118
119   public boolean implies(Permission p) {
120
121     //for (StackTraceElement trace : Thread.currentThread().getStackTrace())
122     //  System.out.println(trace);
123
124     boolean granted = false;
125
126     // Check permission type
127     if (! (p instanceof FridgePermission)) return false;
128     FridgePermission permission = (FridgePermission) p;
129     String target = permission.getName();
130
131     // First try : full match search
132     synchronized (this) {
133       FridgePermission found = (FridgePermission) this.permissions.get(target
              );
134       if (found != null) {
135         int required = found.getBitmask();
136         int effective = permission.getBitmask();
137         granted = ((required & effective) == effective);
138       }
139     }
140
141     // Second try : regexp search
142     if (!granted) synchronized (this) {
143       for(Object key : this.permissions.keySet().toArray()) {
144         if (target.matches((String)key)) {
145           FridgePermission found = (FridgePermission) this.permissions.get(
                  key);
146           int required = found.getBitmask();
147           int effective = permission.getBitmask();
148           granted = ((required & effective) == effective);
149
150           // We found a granted permission
151           if (granted) break;
152         }
153       }
154     }
155
156     //System.out.println("granted ==> " + granted);
157     return granted;
158   }
159
160   public Enumeration elements() {
161     // Convert Iterator of Map values into an Enumeration
162     synchronized (this) {
163       return Collections.enumeration(this.permissions.values());
164     }
```

```
165   }
166 }
```

Listing B.5 – Code source de la permission pour accéder au Réfrigérateur

```
1 grant codeBase "file:./trusted.jar" {
2   permission FridgePermission ".*", "take,put";
3 };
4
5 grant codeBase "file:./not-trusted.jar" {
6   permission FridgePermission ".*", "put";
7 //  permission FridgePermission "Soda", "take";
8 //  permission FridgePermission "Beer", "take";
9 };
```

Listing B.6 – Politique JAAS du cas d'étude

## Classe principale qui implémente le scénario

La classe *Usecase* configure les 6 scénarii présentés dans [83]. Le choix de l'un d'entre eux se fait en passant un nombre compris entre 1 et 6 au script de lancement :

1. La **fille** prend un jus d'orange au **réfrigérateur**
2. Le **maçon** prend une bière au **réfrigérateur**
3. Le **maçon** demande à la **fille** de prendre une bière au **réfrigérateur**
4. La **fille** demande au **maçon** de prendre un jus d'orange au **réfrigérateur**
5. Le **maçon** demande à la **fille** de prendre une bière au **réfrigérateur** de façon privilégiée
6. La **fille** demande au **maçon** de prendre un jus d'orange au **réfrigérateur** de façon privilégiée

```
1  import java.io.PrintStream;
2  import java.io.UnsupportedEncodingException;
3
4  // Availabe drinks
5  class Drink {  }
6  class Beer extends Drink {  }
7  class Soda extends Drink {  }
8  class Juice extends Drink {  }
9
10 // Usecase implementation
11 public class Usecase {
12
13   public static int scenario = 0;
14
15   public static void main(String[] argv) {
16
17     // Parse command line arguments to retrieve scenario ID
18     if (argv.length > 0) try {
19       scenario = Integer.parseInt(argv[0]);
20     } catch(NumberFormatException e) { }
21
22     // Prepare entities
```

```
23      Fridge fridge = new Fridge();
24      Builder jean_claude = new Builder(fridge);
25      Daughter pierrette = new Daughter(fridge);
26
27      // ===== Usecase Statement =====
28      System.out.println(" Cas d'étude :\n -------------\n");
29      System.out.println("  Pendant mon absence un maçon est occupé à effectuer
            des travaux.");
30      System.out.println("  Ma fille est en congé et à la maison au même moment
            .");
31      System.out.println("  Le maçon n'a pas la permision d'accéder au frigo
            tandis que ma fille bien.");
32      System.out.println("\n  (@see http://wikilabus.steformations.be/wikilabus
            /index.php/Contrôle_d%27accès_en_Java)");
33
34      // ===== Active Policy =====
35      System.out.println("\n Contextes de sécurités :\n
            ------------------------\n");
36      System.out.println("  Fille ==> " + Daughter.class.getProtectionDomain())
            ;
37      System.out.println("  Maçon ==> " + Builder.class.getProtectionDomain());
38
39      // ===== Scenario Statement =====
40      System.out.println("\n Scénario :\n ----------\n");
41      Object drink = null;
42      switch (scenario) {
43        default: scenario = 1;
44        case 1: { // ===== Scenario #1 =====
45          System.out.println("  Ma fille demande un jus d'orange au frigo.\n");
46
47          // Put some drinks in fridge
48          fridge.put(new Juice());
49
50          // Retrieve a drink from fridge
51          drink = pierrette.take();
52        } break;
53        case 2: { // ===== Scenario #2 =====
54          System.out.println("  Le maçon demande une bière au frigo.\n");
55
56          // Put some drinks in fridge
57          fridge.put(new Beer());
58
59          // Retrieve a drink from fridge
60          drink = jean_claude.take();
61        } break;
62        case 5: System.out.print("  (privileged)");
63        case 3: { // ===== Scenario #3 =====
64          System.out.println("  Le maçon demande à ma fille de prendre une
                bière dans le frigo.\n");
65
66          // Put some drinks in fridge
67          fridge.put(new Beer());
68
69          // Retrieve a drink from fridge
70          drink = jean_claude.request(pierrette);
71        } break;
72        case 6: System.out.print("  (privileged)");
73        case 4: { // ===== Scenario #4 =====
74          System.out.println("  Ma fille demande au maçon de prendre un jus d'
                orange dans le frigo.\n");
75
76          // Put some drinks in fridge
77          fridge.put(new Juice());
78
79          // Retrieve a drink from fridge
80          drink = pierrette.request(jean_claude);
81        } break;
82      }
83
84      // Scenario ends
85      System.out.println(" Boisson obtenue :\n -----------------\n\n  " + drink
            .getClass().getName());
86  }
```

```
87 }
```

Listing B.7 – Code source de la classe principale

## Scripts pour compiler et lancer les scénarii du cas d'étude

```
1  @echo off
2
3  rem compiling
4  chcp 65001 && javac *.java %1
5
6  rem packing
7  for /R . %%F in (*.class) do (
8    set p=%%F
9    setlocal EnableDelayedExpansion
10   if "!p:%__CD__%=!"=="Builder.class" (echo !p:%__CD__%=! >> not-trusted.txt)
11   if not "!p:%__CD__%=!"=="Builder.class" (echo !p:%__CD__%=! >> trusted.txt)
12   endlocal
13 )
14 jar cvf not-trusted.jar @not-trusted.txt
15 jar cvf trusted.jar @trusted.txt
16
17 rem cleaning
18 del *.class
19 del trusted.txt
20 del not-trusted.txt
```

Listing B.8 – Script Windows pour compiler le cas d'étude

```
1  @echo off
2
3  rem script's argument requires a scenario number
4  if %1.==. (
5    echo usage :
6    echo       %~n0 ^<scenario^>
7    echo.
8    echo scenario :
9    echo       1: girl --^{ take^(^) ^}--^> fridge
10   echo       2: builder --^{ take^(^) ^}--^> fridge
11   echo       3: builder --^{ request^(^) ^}--^> girl --^{ take^(^) ^}--^>
         fridge
12   echo       4: girl --^{ request^(^) ^}--^> builder --^{ take^(^) ^}--^>
         fridge
13   echo       5: builder --^{ request^(^) ^}--^> girl --^{ doPrivileged^(take
         ^(^)^) ^}--^> fridge
14   echo       6: girl --^{ request^(^) ^}--^> builder --^{ doPrivileged^(take
         ^(^)^) ^}--^> fridge
15   goto:eof)
16
17 rem launch usecase by forcing UTF8 enconding
18 chcp 65001 && java -cp "trusted.jar;not-trusted.jar;." -Dfile.encoding=UTF-8
       -Djava.security.manager -Djava.security.policy=usecase.policy Usecase %1
```

Listing B.9 – Script Windows pour exécuter le cas d'étude

Annexe C

# Index du modèle général

## Table des hypothèses



## Table des notations

## Table des suggestions

# Bibliographie


[1] W. P. Rogers, N. Armstrong, S. Ride, D. Acheson, E. Covert, R. Hotz, R. Feynman, A. Wheelon, A. B. C. W. Jr., D. J. Kutyna, R. Rummel, J. Sutter, and C. E. Yeager, "Report of the presidential commission on the space shuttle challenger accident," Independent study, Tech. Rep., june 1986. [Online]. Available : http://science.ksc.nasa.gov/shuttle/missions/51-l/docs/rogers-commission/table-of-contents.html

[2] Patrick Chambet, "La gestion des correctifs de sécurité," *MISC*, no. 22, November 2005. [Online]. Available : http://www.chambet.com/publications/Correctifs_securite.pdf

[3] M. Fonda, "Protection obligatoire des serveurs d'applications web : application aux processus métiers." Ph.D. dissertation, Université d'Orléans, May 2014. [Online]. Available : ftp://ftp.univ-orleans.fr/theses/maxime.fonda_3601.pdf

[4] J. Briffaut, M. Peres, C. Toinard, J. Rouzaud-Cornabas, B. Venelle, and J. Solanki, "PIGA-OS : Retour sur le Système d'Exploitation Vainqueur du Défi Sécurité," in *RenPar'20 / SympA'14 / CFSE 8, 8ème Conférence Francaise en Systèmes d'Exploitation.* Saint-Malo, France : RenPar'20 / SympA'14 / CFSE 8, May 2011. [Online]. Available : http://hal.inria.fr/hal-00671591

[5] E. W. Dijkstra, "Computing science : Achievements and challenges," in *ACM SIGAPP Applied Computing Review*, vol. 7, no. 2. ACM, 1999, pp. 2–9. [Online]. Available : http://www.cs.utexas.edu/users/EWD/transcriptions/EWD12xx/EWD1284.html

[6] Robert Jeffries, "Java and security : 17 years in (brief) review," Solutionary, Tech. Rep., march 2013. [Online]. Available : http://www.solutionary.com/resource-center/blog/2013/03/java-and-security/

[7] S. Govindavajhala and A. W. Appel, "Using memory errors to attack a virtual machine," in *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, ser. SP '03. Washington, DC, USA : IEEE Computer Society, 2003, pp. 154–. [Online]. Available : http://dl.acm.org/citation.cfm?id=829515.830563

[8] L. S. of Delirium Research Group, "Java and java virtual machine security vulnerabilities and their exploitation techniques," in *Proceedings of Black Hat Asia 2002 Singapore*, 2002. [Online]. Available : https://www.blackhat.com/presentations/bh-asia-02/LSD/bh-asia-02-lsd-article.pdf

[9] Jeong Wook (Matt) Oh, "Recent java exploitation trends and malware," in *Proceedings of Black Hat USA 2012 Las Vegas*, 2012. [Online]. Available : https://media.blackhat.com/bh-us-12/Briefings/Oh/BH_US_12_Oh_Recent_Java_Exploitation_Trends_and_Malware_WP.pdf

[10] Consortium JAVASEC, "Sécurité et langage java," Agence Nationale de la Sécurité des Systèmes d'Information (ANSSI), Tech. Rep., may 2010. [Online]. Available : http://www.ssi.gouv.fr/fr/anssi/publications/publications-scientifiques/autres-publications/securite-et-langage-java.html

[11] A. Gowdiak, "Security vulnerabilities in java se," Security Explorations, Poland, Tech. Rep., November 2012. [Online]. Available : http://www.security-explorations.com/materials/se-2012-01-report.pdf

[12] Sun Microsystems Inc, *Secure Coding Guidelines for Java SE*, Oracle America Inc, april 2014. [Online]. Available : http://www.oracle.com/technetwork/java/seccodeguide-139067.html

[13] G. B. Tim Lindholm, Frank Yellin and A. Buckley, *The Java Virtual Machine Specification, Java SE 7 Edition*, Oracle America Inc, july 2011. [Online]. Available : http://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf

[14] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley, *The Java Language Specification, Java SE 7 Edition*, Oracle America Inc, july 2011. [Online]. Available : http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf

[15] H.-S. Oh, B.-J. Kim, H.-K. Choi, and S.-M. Moon, "Evaluation of android dalvik virtual machine," in *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, ser. JTRES '12. New York, NY, USA : ACM, 2012, pp. 115–124. [Online]. Available : http://doi.acm.org/10.1145/2388936.2388956

[16] B. Venners, *Inside the Java Virtual Machine*, 2nd ed. McGraw-Hill Professional, 1999.

[17] H. M. James Gosling, *The Java Language Environment : Contents*. Sun Microsystems, may 1996. [Online]. Available : http://www.oracle.com/technetwork/java/langenv-140151.html

[18] Sun Microsystems Inc, *Java Authentication and Authorization Service (JAAS) Reference Guide*, Oracle America Inc, 2013. [Online]. Available : http://docs.oracle.com/javase/7/docs/technotes/guides/security/jaas/JAASRefGuide.html

[19] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," 1975.

[20] Sun Microsystems Inc, *Permissions in Java SE 7 Development Kit (JDK)*, Oracle America Inc, 2013. [Online]. Available : http://docs.oracle.com/javase/7/docs/technotes/guides/security/permissions.html

[21] J. P. Anderson, "Information security in a multi-user computer environment," *Advances in Computers*, vol. 12, pp. 1–36, 1972.

[22] R. Scheifler and L. Gong, "Stack based access control using code and executor identifiers," May 14 2002, uS Patent 6,389,540. [Online]. Available : http://www.google.com/patents/US6389540

[23] D. S. Wallach and E. W. Felten, "Understanding java stack inspection," in *In Proceedings of the 1998 IEEE Symposium on Security and Privacy*, 1998, pp. 52–63.

[24] Microsoft, "Code access security in .net framework 4.5," Microsoft Developer Network (MSDN), Tech. Rep., 2014. [Online]. Available : http://msdn.microsoft.com/en-us/library/c5tk9z76(v=vs.110).aspx

[25] F. Long, D. Mohindra, R. C. Seacord, D. F. Sutherland, and D. Svoboda, *The CERT Oracle secure coding standard for Java*, ser. The SEI series in software engineering. Upper Saddle River, NJ : Addison-Wesley, 2012. [Online]. Available : https://www.securecoding.cert.org/confluence/display/java/The+CERT+Oracle+Coding+Standard+for+Java

[26] M. Abadi and C. Fournet, "Access control based on execution history," in *In Proceedings of the 10th Annual Network and Distributed System Security Symposium*, 2003, pp. 107–121.

[27] M. Pistoia, "Beyond stack inspection : A unified access-control and information-flow security model," in *In SP'07 : Security and Privacy*. IEEE, 2007, pp. 149–163.

[28] S. Zdancewic, "A type system for robust declassification," *Electronic Notes in Theoretical Computer Science*, vol. 83, pp. 263–277, 2013.

[29] X. Leroy, "Java bytecode verification : Algorithms and formalizations," *J. Autom. Reason.*, vol. 30, no. 3-4, pp. 235–269, Aug. 2003. [Online]. Available : http://dx.doi.org/10.1023/A:1025055424017

[30] H. G. Rice, "Classes of recursively enumerable sets and their decision problems," *Trans. Amer. Math. Soc.*, vol. 74, pp. 358–366, 1953.

[31] ISO, "Iso c standard 1999," ISO/IEC, Tech. Rep., 1999, iSO/IEC 9899 :1999 draft. [Online]. Available : http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf

[32] S. Nair, P. Simpson, B. Crispo, and A. Tanenbaum, "Trishul : A policy enforcement architecture for java virtual machines," in *Technical Report IR-CS-045*, may 2008.

[33] J. Clause, W. Li, and A. Orso, "Dytan : A generic dynamic taint analysis framework," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ser. ISSTA '07. New York, NY, USA : ACM, 2007, pp. 196–206. [Online]. Available : http://doi.acm.org/10.1145/1273463.1273490

[34] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP '10. Washington, DC, USA : IEEE Computer Society, 2010, pp. 317–331. [Online]. Available : http://dx.doi.org/10.1109/SP.2010.26

[35] G. Hiet, V. V. T. Tong, L. Me, and B. Morin, "Policy&#45 ;based intrusion detection in web applications by monitoring java information flows," *Int. J. Inf. Comput. Secur.*, vol. 3, no. 3/4, pp. 265–279, Jan. 2009.

[36] R. Andriatsimandefitra, S. Geller, and V. Tong, "Designing information flow policies for android's operating system," in *Communications (ICC), 2012 IEEE International Conference on*, June 2012, pp. 976–981.

[37] R. Andriatsimandefitra, V. Viet Triem Tong, and L. Mé, "Diagnosing intrusions in Android operating system using system flow graph," in *Workshop Interdisciplinaire sur la Sécurité Globale*, Troyes, France, Jan. 2013. [Online]. Available : http://hal.inria.fr/hal-00875211

[38] M. Jaume, V. Viet Triem Tong, and L. Mé, "Flow based interpretation of access control : Detection of illegal information flows," in *Proceedings of the 7th International Conference on Information Systems Security (ICISS)*, vol. 7093, Kolkata, Inde, Dec. 2011, pp. 72–86. [Online]. Available : http://hal-supelec.archives-ouvertes.fr/hal-00647170

[39] G. Hiet, L. Mé, B. Morin, V. V. T. Tong *et al.*, "Monitoring both os and program level information flows to detect intrusions against network servers," *Proceedings of IEEE Workshop on Monitoring, Attack Detection and Mitigation*, 2007.

[40] T. Letan, "Proposition et implémentation d'une coopération entre deux moniteurs de flux d'information," Master's thesis, INRIA-IRISA Rennes Bretagne Atlantique, équipe CIDRE, Jun. 2013. [Online]. Available : http://dumas.ccsd.cnrs.fr/dumas-00854981

[41] V. Haldar, D. Chandra, and M. Franz, "Dynamic taint propagation for java," Department of Information and Computer Science - University of California, Tech. Rep., 2005.

[42] ——, "Practical, dynamic information-flow for virtual machines," Department of Information and Computer Science - University of California, Tech. Rep., September 2005.

[43] L. Cavallaro, P. Saxena, and R. Sekar, "On the limits of information flow techniques for malware analysis and containment," in *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. DIMVA '08. Berlin, Heidelberg : Springer-Verlag, 2008, pp. 143–163. [Online]. Available : http://dx.doi.org/10.1007/978-3-540-70542-0_8

[44] J. Andronick, B. Chetali, and C. Paulin-Mohring, "Formal verification of security properties of smart card embedded source code," in *Proceedings of the 2005 International Conference on Formal Methods*, ser. FM'05. Berlin, Heidelberg : Springer-Verlag, 2005, pp. 302–317. [Online]. Available : http://dx.doi.org/10.1007/11526841_21

[45] P. Li and S. Zdancewic, "Arrows for secure information flow," *Theor. Comput. Sci.*, vol. 411, no. 19, pp. 1974–1994, Apr. 2010. [Online]. Available : http://dx.doi.org/10.1016/j.tcs.2010.01.025

[46] V. Simonet and I. Rocquencourt, "Flow caml in a nutshell," in *Proceedings of the first APPSEM-II workshop*, 2003, pp. 152–165.

[47] A. C. Myers, S. Chong, N. Nystrom, L. Zheng, and S. Zdancewic, "Jif : Java Information Flow," located at http ://www.cs.cornell.edu/jif.

[48] Legifrance, Ed., *Code de la propriété intellectuelle : (partie législative)*. Legifrance, july 2014. [Online]. Available : http://www.legifrance.gouv.fr/affichCode.do?cidTexte=LEGITEXT000006069414

[49] F. Pottier and V. Simonet, "Information flow inference for ml," *ACM Trans. Program. Lang. Syst.*, vol. 25, no. 1, pp. 117–158, Jan. 2003. [Online]. Available : http://doi.acm.org/10.1145/596980.596983

[50] A. Askarov and A. C. Myers, "Attacker control and impact for confidentiality and integrity," *Logical Methods in Computer Science*, vol. 7, no. 3, 2011.

[51] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman, "Protection in operating systems," *Commun. ACM*, vol. 19, no. 8, pp. 461–471, Aug. 1976. [Online]. Available : http://doi.acm.org/10.1145/360303.360333

[52] E. D. Bell and J. L. La Padula, "Secure computer system : Unified exposition and multics interpretation," Bedford, MA, 1976. [Online]. Available : http://csrc.nist.gov/publications/history/bell76.pdf

[53] D. E. Bell, "Looking back at the bell-la padula model," in *Proceedings of the 21st Annual Computer Security Applications Conference*, ser. ACSAC '05. Washington, DC, USA : IEEE Computer Society, 2005, pp. 337–351. [Online]. Available : http://dx.doi.org/10.1109/CSAC.2005.37

[54] K. J. Biba, "Integrity considerations for secure computer systems," MITRE corporation, Tech. Rep., 1977.

[55] Microsoft, "Windows vista integrity mechanism technical reference," Microsoft Developer Network (MSDN), Tech. Rep., 2006. [Online]. Available : http://msdn.microsoft.com/en-us/library/bb625964.aspx

[56] B. Hicks, S. Rueda, T. Jaeger, and P. Mcdaniel, "From trusted to secure : Building and executing applications that enforce system security," in *USENIX ANNUAL TECHNICAL CONFERENCE*. Berkeley, CA, USA, 2007. [Online]. Available : https://www.usenix.org/legacy/event/usenix07/tech/full_papers/hicks/hicks.pdf

[57] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel, "Laminar : practical fine-grained decentralized information flow control," in *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '09. New York, NY, USA : ACM, 2009, pp. 63–74. [Online]. Available : http://doi.acm.org/10.1145/1542476.1542484

[58] A. C. Myers and B. Liskov, "Protecting privacy using the decentralized label model," *ACM Trans. Softw. Eng. Methodol.*, vol. 9, no. 4, pp. 410–442, Oct. 2000. [Online]. Available : http://doi.acm.org/10.1145/363516.363526

[59] V. Simonet and I. Rocquencourt, "Flow caml in a nutshell," in *Proceedings of the first APPSEM-II workshop*, 2003, pp. 152–165.

[60] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris, "Information flow control for standard os abstractions," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 321–334, Oct. 2007. [Online]. Available : http://doi.acm.org/10.1145/1323293.1294293

[61] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August, "Rifle : An architectural framework for user-centric information-flow security." in *MICRO*. IEEE Computer Society, 2004, pp. 243–254.

[Online]. Available : http://dblp.uni-trier.de/db/conf/micro/micro2004.html#VachharajaniBCROBRVA04

[62] P. Loscocco and S. Smalley, "Integrating flexible support for security policies into the linux operating system," in *Proceedings of the FREENIX Track : 2001 USENIX Annual Technical Conference*. Berkeley, CA, USA : USENIX Association, 2001, pp. 29–42. [Online]. Available : http://dl.acm.org/citation.cfm?id=647054.715771

[63] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat, "A domain and type enforcement unix prototype," in *In Proceedings of the Fifth USENIX UNIX Security Symposium*, 1996, pp. 127–140.

[64] M. R. Azadmanesh and M. Sharifi, "Towards a system-wide and transparent security mechanism using language-level information flow control," in *Proceedings of the 3rd international conference on Security of information and networks*, ser. SIN '10. New York, NY, USA : ACM, 2010, pp. 19–26. [Online]. Available : http://doi.acm.org/10.1145/1854099.1854107

[65] S. Smalley, "Middleware mac for seandroid," *Linux security submit*, august 2012. [Online]. Available : http://selinuxproject.org/page/SEandroid

[66] J. Briffaut, M. Peres, and C. Toinard, "A dynamic end-to-end security for coordinating multiple protections within a linux desktop." in *CTS*, W. W. Smari and W. K. McQuay, Eds. IEEE, 2010, pp. 509–515. [Online]. Available : http://dblp.uni-trier.de/db/conf/cts/cts2010.html#BriffautPT10

[67] M. Fonda, S. Moinard, and C. Toinard, "Advanced protection of workflow sessions with sewebsession." in *Business Process Management Workshops*, ser. Lecture Notes in Business Information Processing, M. L. Rosa and P. Soffer, Eds., vol. 132. Springer, 2012, pp. 713–718. [Online]. Available : http://dblp.uni-trier.de/db/conf/bpm/bpmw2012.html#FondaMT12

[68] A. Kay, "Prototypes vs classes," Squeak mailling list, October 1998. [Online]. Available : http://lists.squeakfoundation.org/pipermail/squeak-dev/1998-October/017019.html

[69] O. Dahl, "The birth of object orientation : the simula languages," in *From Object-Orientation to Formal Methods, Essays in Memory of Ole-Johan Dahl*, 2004, pp. 15–25. [Online]. Available : http://dx.doi.org/10.1007/978-3-540-39993-3_3

[70] P.-A. Muller and N. Gaertner, *Modélisation objet avec UML*, 2nd ed. Eyrolles, March 2000.

[71] O. M. Group, *OMG Unified Modeling LanguageTM (OMG UML), Infrastructure*, Object Management Group, august 2011. [Online]. Available : http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF

[72] ——, *OMG Unified Modeling LanguageTM (OMG UML), Superstructure*, Object Management Group, august 2011. [Online]. Available : http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF

[73] R. Colvin, "An operational semantics for object-oriented concepts based on the class hierarchy," *Formal Aspects of Computing*, vol. 26, no. 3, pp. 491–535, 2014. [Online]. Available : http://dx.doi.org/10.1007/s00165-012-0259-y

[74] A. Hense, "Denotational semantics of an object-oriented programming language with explicit wrappers," *Formal Aspects of Computing*, vol. 5, no. 3, pp. 181–207, 1993. [Online]. Available : http://dx.doi.org/10.1007/BF01211554

[75] M. Abadi and L. Cardelli, *A Theory of Objects*, 1st ed. Secaucus, NJ, USA : Springer-Verlag New York, Inc., 1996.

[76] A. M. Turing, "Computers & thought," in *Mind*, E. A. Feigenbaum and J. Feldman, Eds. Cambridge, MA, USA : MIT Press, 1995, vol. 59, ch. Computing Machinery and Intelligence, pp. 11–35. [Online]. Available : http://dl.acm.org/citation.cfm?id=216408.216410

[77] G. B. Tim Lindholm, Frank Yellin and A. Buckley, *The Java Virtual Machine Specification, Java SE 8 Edition*, Oracle America Inc, march 2014. [Online]. Available : http://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf

[78] J. Rouzaud-Cornabas, "Formalisation de propriétés de sécurité pour la protection des systèmes d'exploitation," Ph.D. dissertation, Université d'Orléans, Dec 2010. [Online]. Available : http://tel.archives-ouvertes.fr/tel-00623075/PDF/jonathan.rouzaudcornabas_2180_vm.pdf

[79] J. Briffaut, "Formalisation et garantie de propriétés de sécurité système : application á la détection d'intrusions," Ph.D. dissertation, Université d'Orléans, Dec 2007. [Online]. Available : http://tel.archives-ouvertes.fr/tel-00261613/PDF/thesis.pdf

[80] A. Bousquet, J. Briffaut, L. Clévy, C. Toinard, and B. Venelle, "Mandatory Access Control for the Android Dalvik Virtual Machine," in *2013 - USENIX Federated Conferences, ESOS : Workshop on Embedded Self-Organizing Systems*, San Jose, États-Unis, Jun. 2013. [Online]. Available : http://hal.inria.fr/hal-00840732

[81] B. Venelle, J. Briffaut, L. Clévy, and C. Toinard, "Security Enhanced Java : Mandatory Access Control for the Java Virtual Machine," in *ISORC - 6h IEEE International Symposium on Object, Component, and Service-Oriented Real-Time Distributed Computing - 2013*, Paderborn, Allemagne, Jun. 2013. [Online]. Available : http://hal.inria.fr/hal-00840729

[82] E. Snowden, "Edward snowden in his own words," in *PRISM Whistleblower*, 2014. [Online]. Available : http://www.youtube.com/watch?v=3P_0iaCgKLk

[83] S. formations, "Contrôle d'accès en java." [Online]. Available : http://wikilabus.steformations.be/wikilabus/index.php/Contrôle_d'accès_en_Java

[84] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, Jul. 1982. [Online]. Available : http://doi.acm.org/10.1145/357172.357176

[85] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, Apr. 1985. [Online]. Available : http://doi.acm.org/10.1145/3149.214121

[86] G. Hunt and D. Brubacher, "Detours : Binary interception of win32 functions," in *Proceedings of the 3rd Conference on USENIX*

*Windows NT Symposium - Volume 3*, ser. WINSYM'99. Berkeley, CA, USA : USENIX Association, 1999, pp. 14–14. [Online]. Available : http://dl.acm.org/citation.cfm?id=1268427.1268441

[87] Michael 'mihi' Schierl, "Cve-2012-1723 openjdk : insufficient field accessibility checks," Independent study, Tech. Rep., 2012. [Online]. Available : http://schierlm.users.sourceforge.net/CVE-2012-1723.html

[88] Sami Koivu, "Cve-2008-5353 : Calendar bug," (Slightly) Random Broken Thoughts, Tech. Rep., december 2008. [Online]. Available : http://slightlyrandombrokenthoughts.blogspot.fr/2008/12/calendar-bug.html

[89] Esteban Guillardoy, "Java 0day analysis (cve-2012-4681)," Independent study, Tech. Rep., 2012. [Online]. Available : http://immunityproducts.blogspot.fr/2012/08/java-0day-analysis-cve-2012-4681.html

[90] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat, "Practical domain and type enforcement for unix," in *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, ser. SP '95. Washington, DC, USA : IEEE Computer Society, 1995, pp. 66–. [Online]. Available : http://dl.acm.org/citation.cfm?id=882491.884237

[91] ISO, "Iso c++ standard 2011," ISO/IEC, Tech. Rep., 2011, iSO/IEC 14882 :2011. [Online]. Available : http://www.open-std.org/jtc1/sc22/wg21/

[92] Sun Microsystems Inc, *Default Policy Implementation and Policy File Syntax*, Oracle America Inc, 2013. [Online]. Available : http://docs.oracle.com/javase/7/docs/technotes/guides/security/PolicyFiles.html

[93] ——, *Using applet, object and embed Tags*, Oracle America Inc, 2014. [Online]. Available : http://docs.oracle.com/javase/7/docs/technotes/guides/jweb/applet/using_tags.html

[94] P. Seibel, *Coders at Work : Reflections on the Craft of Programming*, ser. Apresspod Series. Apress, 2009. [Online]. Available : http://books.google.fr/books?id=nneBa6-mWfgC

[95] t. f. e. Wikipedia, "Comparison of java virtual machines," Wikipedia's article, may 2015. [Online]. Available : http://en.wikipedia.org/wiki/Comparison_of_Java_virtual_machines

[96] Oracle, *The Java Native Interface Specification, Java SE 7 Edition*, Oracle America Inc, july 2014. [Online]. Available : http://docs.oracle.com/javase/7/docs/technotes/guides/jni/

[97] Oracle, *The Java Virtual Machine Tool Interface Specifications*, Oracle America Inc, 2012. [Online]. Available : http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html

[98] Sun Microsystems Inc, *The Java Platform Debugger Architecture*, Oracle America Inc, 2004. [Online]. Available : http://docs.oracle.com/javase/1.5.0/docs/guide/jpda/architecture.html

[99] C.K Prasad, Rajesh Ramchandani, Gopinath Rao, and Kim Levesque, *Creating a Debugging and Profiling Agent with JVMTI*, Sun Microsystems Inc, june 2004. [Online]. Available : http://www.oracle.com/technetwork/articles/java/jvmti-136367.html

[100] Robert Field, *JVM Tool Interface, Implementation in Hotspot*, Sun Microsystems Inc, june 2007. [Online]. Available : http://openjdk.java.net/groups/hotspot/docs/jvmtiImpl.pdf

[101] S. Chiba and M. Nishizawa, "An easy-to-use toolkit for efficient java bytecode translators," in *Proceedings of the 2Nd International Conference on Generative Programming and Component Engineering*, ser. GPCE '03. New York, NY, USA : Springer-Verlag New York, Inc., 2003, pp. 364–376. [Online]. Available : http://dl.acm.org/citation.cfm?id=954186.954208

[102] Thomas Queste, *An introduction to Java Agent and bytecode manipulation*, january 2014. [Online]. Available : http://www.tomsquest.com/blog/2014/01/intro-java-agent-and-bytecode-manipulation/

[103] Oracle, *The Java Virtual Machine Tool Interface Specifications - Bytecode Instrumentation*, Oracle America Inc, 2012. [Online]. Available : http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#bci

[104] Muhammad Khojaye, *Getter Setter : To Use or Not to Use*, october 2010. [Online]. Available : http://java.dzone.com/articles/getter-setter-use-or-not-use-0

[105] Florian Lugou, "Instrumentation de machine virtuelle Java orientée contrôle d'accès obligatoire," Master's thesis, EURECOM, September 2014.

[106] R. Spencer, S. Smalley, P. Loscocco, P. L. (national Security Agency, M. Hibler, J. Lepreau, and D. Andersen, "The flask security architecture : System support for diverse security policies," in *in Proceedings of The Eighth USENIX Security Symposium*, 1998, pp. 123–139.

[107] B. Stroustrup, *The C++ Programming Language*, 3rd ed. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2000. [Online]. Available : http://www.stroustrup.com/bs_faq.html#really-say-that

[108] C. Toinard, "Towards an autonomous and distributed security management of federated clouds," in *Proceedings of the 2015 Cloud Security Workshop (SEC2 - Lille)*. INRIA, june 2015.

**Benjamin VENELLE**

# Contrôle d'accès obligatoire pour systèmes à objets
# Défense en profondeur des objets Java

Les systèmes à objets sont présents partout dans notre quotidien. Ainsi, une vulnérabilité dans ces systèmes compromet amplement la confidentialité ou l'intégrité. Par exemple, Java est un système à objets basé sur les classes qui a été la cible de nombreuses cyber-attaques entre 2012 et 2013 au point que le département de la sécurité intérieure des Etats-Unis recommande son abandon.

Dans cette thèse nous proposons de limiter les relations entre objets au moyen d'un contrôle d'accès obligatoire. Pour cela nous définissons un modèle général de système à objets supportant notamment les langages objets et à prototypes. Puis nous formalisons les relations élémentaires que nous pouvons observer et donc contrôler. Celles-ci sont la référence, l'interaction et trois types de flux (d'activité, d'information, de données). Nous proposons également une logique basée sur des automates qui permet de calculer les privilèges de chaque objet. Ainsi, nous calculons dynamiquement la politique obligatoire nécessaire pour satisfaire les objectifs de sécurité exigés. Par là même, nous résolvons d'un seul coup le calcul des politiques obligatoires et le problème d'efficacité puisque la politique obligatoire se trouve réduite. L'expérimentation propose une application aux objectifs de sécurité JAAS existants dans Java. De fait, nous avons été capables d'empêcher les malwares Java correspondant à une année de vulnérabilités au moyen de l'outil d'exploitation Metasploit.

Mots clés : Sécurité, Contrôle d'accès obligatoire, Autonomie, Système à objets, Java.

# Mandatory access control for object systems
# Defense in depth for Java objects

Objects based systems are presents everywhere in our life. When such a system presents vulnerabilities, confidentiality and integrity are thus widely compromised. For example, Java is an object language authorizing many cyber-attacks between 2012 and 2013 leading the US department of homeland security to recommend its abandon.

This thesis proposes to limit the relations between the objects thanks to a mandatory access control. First, a general model of objects supporting objects and prototypes languages is defined. Second, the elementary relations are formalized in order to control them. Those relations include the reference, interaction and three types of flow (activity, information and data). Automata authorize a logic that enables to compute the required mandatory policy. At the same time, the computation of the MAC policy and the efficiency are solved since the policy is reduced. Experimentations use the JAAS security objectives existing in the Java language. Thus, one year of Java vulnerabilities is prevented thanks to the Metasploit framework.

Keywords: Security, Mandatory Access Control, Autonomous, Objects based systems, Java.

LABORATOIRE D'INFORMATIQUE FONDAMENTALE D'ORLEANS

**Laboratoire d'Informatique Fondamentale d'Orléans**

**Bâtiment IIIA Rue Léonard de Vinci**

**BP 6759 F – 45067 ORLEANS Cedex 2**

Alcatel·Lucent

**Alcatel-Lucent Bell Labs France**

**Centre de Villarceaux**

**Route de Villejust – 91460 Nozay**